Electronic Notes in Theoretical Computer Science
Volume 4

Rewriting Logic and its Applications
First International Workshop

Asilomar Conference Center, Pacific Grove, CA
September 3 - 6, 1996

Guest Editor:
J. Meseguer

DTIC QUALITY INSPECTED 4

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

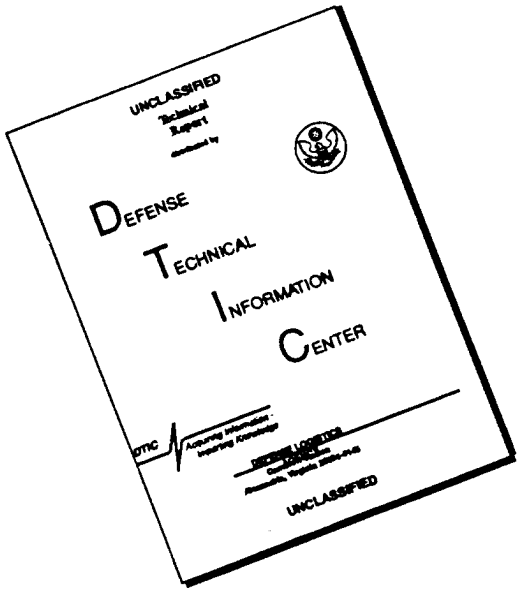
Sponsors:

SRI International
Office of Naval Research
National Security Agency

© Elsevier Science B.V.

19960930 082

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searc gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate o collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations a Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washi

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	9/6/96	Proceedings Sept. 3 through Sept. 6, 1007

4. TITLE AND SUBTITLE
Rewriting LOgic and its Applications, First International Workshop, Proceedings

5. FUNDING NUMBERS
ONR Grant N00014-96-1-0824

6. AUTHOR(S)
J. Meseguer, Editor

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

8. PERFORMING ORGANIZATION REPORT NUMBER
ECU 7242

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Program Officer Ralph F. Wachter, Code 311
800 North Quincy St., Arlington, VA 22217-5660

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Published by Elsevier Science B.V. in Electronic Notes in Theoretical Computer Science, Vol. 4 Copyright Elsevier B.V.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This volume constitutes the proceedings of the First International Workshop on Rewriting Logic, held at the Asilomar Conference Center, Pacific Grove, California, Sept. 2-6, 1996. The three invited papers by Narciso Marti-Oliet, Ugo Montanari, and Martin Wirsing, and seventeen contributed papers selected by the Program Committee give a rich view of the latest developments and research directions in the field of rewriting logic and its various applications to computing. Besides work on models and on concurrency aspects, there are several papers describing the different rewriting logic languages developed so far in Europe and the U.S., as well as a paper on semantic foundations for the Cafe language in Japan. There are also several papers on logical and metalogical specification; on reflection and strategies; on applications to object-oriented design, specification and programming; and on applications to contraint solving, to real-time systems, and to discrete event simulation. In addition to the papers, the workshop placed strong emphasis on facilitating in-depth discussion of the topics by allowing ample time for the discussion of each paper, and by including five panel discussions and a tutorial.

14. SUBJECT TERMS
Rewriting Logic

15. NUMBER OF PAGES
426

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Preface

This volume constitutes the proceedings of the First International Workshop on Rewriting Logic and its Applications, held at the Asilomar Conference Center, Pacific Grove, California, September 3–6, 1996.

There are three invited papers by Narciso Marti-Oliet, Ugo Montanari, and Martin Wirsing, and seventeen contributed papers selected by the Program Committee among the submitted papers. They give a rich view of the latest developments and research directions in the field of rewriting logic and its different applications to computing. Besides work on models and on concurrency aspects, there are several papers describing the different rewriting logic languages developed so far in Europe and the U.S. , as well as a paper on semantic foundations for the Cafe language in Japan. There are also several papers on logical and metalogical specification; on reflection and strategies; on applications to object-oriented design, specification and programming; and on applications to constraint solving, to real-time systems, and to discrete event simulation.

Besides the papers, the workshop's program has placed strong emphasis on facilitating in-depth discussion of the topics by allowing ample time for the discussion of each paper, by including five panel discussions on specific topics, and by scheduling a tutorial at the beginning of the workshop. I wish to thank all the panelists for having contributed to making the workshop a very stimulating scientific dialogue.

I wish to thank my fellow Program Committee members and those who assisted them in the refereeing process. Each submitted paper was refereed by at least three persons. Thanks to their efforts we have a volume of high-quality contributions.

The workshop has been supported by the U.S. Office of Naval Research and by the U.S. National Security Agency through the ONR Grant N00014-96-1-0824. SRI International's Computer Science Laboratory has also contributed resources needed in the preparation of the workshop. All this support is gratefully acknowledged.

Manuel Clavel and Narciso Marti-Oliet have provided invaluable assistance in the preparation of the text of these proceedings. Without them you would not hold this volume in your hands, nor could you browse it on your screen. Judith Burgess deserves very special thanks for her excellent work in all aspects of the workshop's organization.

Menlo Park, California, August 1996 — José Meseguer

Program Committee

Fiorella De Cindio	Università di Milano
Kokichi Futatsugi	Japan Advanced Institute of Science and Technology
Claude Kirchner	INRIA-Lorraine and CRIN-CNRS
Christian Lengauer	Universität Passau
Narciso Martí-Oliet	Universidad Complutense de Madrid
José Meseguer	SRI International (Chair)
Ugo Montanari	Università di Pisa
Carolyn Talcott	Stanford University

Invited Speakers

Narciso Martí-Oliet	Universidad Complutense de Madrid
Ugo Montanari	Università di Pisa
Martin Wirsing	Ludwig-Maximilians-Universität Munchen

Panelists

Hans-Dieter Ehrich	Technische Universität Braunschweig
Joseph Goguen	University of California San Diego
Jean-Pierre Jouannaud	Université Paris-Sud à Orsay

Referees

Peter Borovanský
Carlos Castro
Maura Cerioli
Andrea Corradini
Fabio Gadducci
Gianluigi Ferrari
Hélène Kirchner
François Lamarche
Giancarlo Mauri
Pierre-Etienne Moreau
Francesca Rossi
Nicoletta Sabadini

Table of Contents

Tutorial

A Survey of Rewriting Logic
J. Meseguer (SRI International, USA)

Session on Models

Invited Talk: Tiles, Rewriting Rules, and CCS 1
F. Gadducci and U. Montanari (Università di Pisa, Italy)

Modelling Conditional Rewriting Logic in Structured Categories 20
H. Miyoshi (Shukutoku University, Japan)

Session on Languages

ELAN: A Logical Framework Based on Computational Systems 35
P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek (INRIA Lorraine and CRIN-CNRS, France)

Input/Output for ELAN 51
P. Viry (Università di Pisa, Italy)

Principles of Maude 65
M. Clavel, S. Eker, P. Lincoln, and J. Meseguer (SRI International, USA)

Fast Matching in Combination of Regular Equational Theories 90
S. Eker (SRI International, USA)

Distributed Logic Objects: A Fragment of Rewriting Logic and its Implementation 109
A. Ciampolini, E. Lamma, P. Mello, and C. Stefanelli (Università di Bologna, Italy)

Session on Reflection and Strategies

Reflection and Strategies in Rewriting Logic 125
M. Clavel and J. Meseguer (SRI International, USA)

A Reflective Extension of ELAN 148
H. Kirchner and P.-E. Moreau (INRIA Lorraine and CRIN-CNRS, France)

Controlling Rewriting by Rewriting 168
P. Borovanský, C. Kirchner, and H. Kirchner
(INRIA Lorraine and CRIN-CNRS, France)

Session on Logical and Metalogical Specification

Invited Talk: Rewriting Logic as a Logical
and Semantic Framework 189
N. Martí-Oliet (Universidad Complutense de Madrid, Spain) and
J. Meseguer (SRI International, USA)

Foundations of Behavioural Specification in Rewriting Logic 225
R. Diaconescu (JAIST, Japan)

Solving Binary CSP Using Computational Systems 245
C. Castro (INRIA Lorraine and CRIN, France)

Bi-rewriting Rewriting Logic 265
W. M. Schorlemmer (IIIA-CSIC, Spain)

Session on Real Time and Simulation Applications

Specifying Real-Time Systems in Rewriting Logic 283
P. C. Ölveczky and J. Meseguer (SRI International, USA)

Discrete Event Systems in Rewriting Logic 309
C. Landauer (The Aerospace Corporation, USA)

Session on Object-Oriented Specification and Programming

Invited Talk: A Formal Approach to Object-Oriented
Software Engineering 321
M. Wirsing and A. Knapp
(Ludwig-Maximilians-Universität München, Germany)

An Actor Rewriting Theory 360
C. L. Talcott (Stanford University, USA)

Object-Oriented Specifications of Distributed Systems
in the μ-Calculus and Maude 384
U. Lechner (Universität Passau, Germany)

A Maude Specification of an Object Oriented Database Model
for Telecommunication Networks 404
I. Pita and N. Martí-Oliet
(Universidad Complutense de Madrid, Spain)

Electronic Notes in Theoretical Computer Science 4 (1996)

Tiles, Rewriting Rules and CCS [1]

Fabio Gadducci and Ugo Montanari

Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
({gadducci,ugo}@di.unipi.it*)*

Abstract

In [12] we introduced the *tile model*, a framework encompassing a wide class of computational systems, whose behaviour can be described by certain rewriting rules. We gathered our inspiration both from the world of term rewriting and of concurrency theory, and our formalism recollects many properties of these sources. For example, it provides a compositional way to describe both the states and the sequences of transitions performed by a given system, stressing their distributed nature. Moreover, a suitable notion of typed proof allows to take into account also those formalisms relying on the notions of *synchronization* and *side-effects* to determine the actual behaviour of a system. In this work we narrow our scope, presenting a restricted version of our tile model and focussing our attention on its expressive power. To this aim, we recall the basic definitions of the *process algebras* paradigm [3,24], centering the paper on the recasting of this framework in our formalism.

1 Introduction

It is not an overstatement to say that, in recent years, there has been an unprecedented flow of proposals, aiming at methodologies to describe the semantics of rule-based computational systems. Widely spread in the field of concurrency theory, *transition systems* [16] offered a useful tool for recovering suitable descriptions. They are roughly defined as a set of *states*, representing e.g. the possible memory contents, and a *transition relation* over states, where each element $\langle s, t \rangle$ denotes the evolution from the state s to the state t. Due to its simplicity, however, this view is clearly no more adequate when we need to take into account a compositional structure over states, and the

[1] Research supported in part by Progetto Speciale del CNR "Strumenti per la Specifica e la Verifica di Proprietà Critiche di Sistemi Concorrenti e Distribuiti"; and in part by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

© *1996 Elsevier Science B. V.*

transition relation needs to be inductively defined according to that structure. This is the case of formalisms like *Petri nets* [30], where a state is a multiset of basic components, and each of them may evolve simultaneously (i.e., *in parallel*); or *term rewriting systems* [17], where states are terms of a given algebra, and rewrites are freely obtained from a set of deduction rules. Furthermore, we may need to consider formalisms relying on the use of *synchronization* and *side-effects* in determining the actual behaviour of a system. Maybe, the most important breakthrough is represented by the so-called sos approach [28]: states are compositionally described as terms of a suitable algebra, whose operators express basic features of a system, and the transition relation is defined by means of inference rules, guided by the structure of the states. Along this line, further extensions, which proved fruitful for our view, are *context systems* [22], where the transition relation is defined not on states but on contexts, each of them describing a partially unspecified component of a system; and *structured transition systems* [9,6], where, in order to give a faithful account of the spatial distribution of a system, also transitions are equipped with an algebraic structure.

In [12] we introduced the *tile model*, as an attempt to encompass the properties of the already mentioned formalisms. As it happened for *rewriting logic* [23], the underlying idea of the tile model is to take a logical viewpoint, regarding a rule-based system $\mathcal{R}$ as a logical theory, and any transition step — making use of rules in $\mathcal{R}$ — as a *sequent* entailed by the theory. The entailment relation is defined inductively by a set of *inference rules*, expressing basic features of the model, like its compositional and spatial properties. In particular, there are three composition rules. First, they allow different components of a system to act simultaneously, explicitly describing parallelism by a *monoidal* structure over transitions. Moreover, the compositional structure of states is reflected on computations: sub-components may synchronize and, according to their action, be contextualized. Finally, they can be sequentially composed, expressing in this way the execution of a sequence of transitions.

A sequent $\alpha : s \xrightarrow[b]{a} t$ is a tuple where $s \to t$ is a rewrite step, α is a *proof term* (representing the structure of the step), a is the *trigger* of the step, and b is its *effect*. Its intuitive meaning is: the context s is rewritten to the context t, producing an effect b, but the rule can be applied only if the variables of s (representing still unspecified sub-components) are rewritten with a cumulative effect a. Moreover, two sequents α, β can be composed in parallel ($\alpha \otimes \beta$), composed sequentially ($\alpha \cdot \beta$) or contextualized ($\alpha * \beta$), varying accordingly the corresponding source, target, trigger and effect. Proof terms allow us to equip each rewriting step with a suitable encoding of its causes, while the fact that sequents carry information also about the effect of the associated computation expresses certain *restrictions* about the class of sequents a given rule can be applied to. Alternatively, a sequent can be considered as *synchronized* to its context via its trigger and effect components, and the possibility of expressing restrictions and synchronization will be fundamental when applying our

paradigm to the operational description of *distributed systems.*

The tile model also admits a sofisticated characterization by means of *double-categories* [19], structures that may be roughly described as the superposition of a *vertical* and a *horizontal* category. In [12] it is shown that, starting from a rewriting system, with a free contruction (by means of a suitable adjunction) a double-category can be obtained whose "arrows" are in a one-to-one correspondence with the sequents entailed by that system. This result generalizes the analogous property for term rewriting [29,5], and it underlines the wide applicability of our model [11]. Along this line, in this paper we decided to narrow our attention: instead of describing in full details our formalism, for which we refer the interested reader to [12], we aim at analyzing its expressive power. The focus of the paper, then, is the recasting of the *process algebras* paradigm [3,14,24] in our model, which can be considered as a real benchmark for any general framework (se e.g. [26]). In particular, we deal with a suitable case study, the *Calculus of Communicating Systems* (also CCS, [24]), considered as the standard representative of the paradigm. CCS offers a constructive way to describe *concurrent systems*, considered as structured entities (the *agents*) interacting by means of some synchronization mechanism. Each system is then defined as a term of an algebra over a set of process constructors: new systems are built from existing ones, on the assumption that algebraic operators represent basic features of a concurrent system. The structure over agents allows for an immediate definition of the operational semantics of the language by means of the SOS approach: the dynamic bahaviour of an agent is then described by a suitable *labelled transition system*, where each transition step is a triple $\langle s, \mu, t \rangle$, with μ the observation associated to the transition itself. Finally, a further abstraction is obtained with the associated notion of *bisimulation*: an equivalence over agents equating those with the same observable behaviour.

The paper has the following structure. In Section 2.1 we introduce a formalization of term algebras, providing a concrete description which underlines the assumptions implicitly made in the ordinary notion. In Section 2.2 we introduce our rewriting systems, equipping them with a logic that describes the classes of derivations entailed by a system using (possibly abstract) sequents. In Section 3 we recall the basic definitions of CCS, its operational semantics and the associated strong bisimulation equivalence, along with its finite axiomatization. Finally, in Section 4 we show how the process algebras paradigm can be recovered in our framework. In particular, in Section 4.1 we describe a rewriting system which faithfully recovers the ordinary SOS semantics of CCS; in Section 4.2 we introduce the notion of *tile bisimulation*, in order to recast a suitable notion of observational equivalence in our formalism: this enables us to recover also CCS bisimilarity; finally, in Section 4.3 we turn the finite axiomatization of bisimilarity in a confluent rewriting system, providing each class of bisimilar agents with a canonical representative.

2 A Summary of the Tile Model

In this section we describe the basic features of the *tile model*, within a presentation biased towards the *process algebras* framework we deal with in Sections 3 and 4. For a comprehensive introduction we refer the reader to [12].

2.1 Building States

We open this section recalling some definitions from graph theory, that will be used to introduce *algebraic theories* [21,18]. Developed in the early Sixties, these theories received a lot of attention during the Seventies from computer scientists as a suitable characterization of the ordinary notion of term algebra.

Definition 2.1 (graphs). A *graph* G is a 4-tuple $\langle O_G, A_G, \delta_0, \delta_1 \rangle$: O_G, A_G are sets whose elements are called respectively *objects* and *arrows* (ranged over by $a, b, \ldots$ and $f, g, \ldots$), and $\delta_0, \delta_1 : A_G \to O_G$ are functions, called respectively *source* and *target*. A graph G is *reflexive* when equipped with an *identity* function $id : O_G \to A_G$ such that $\delta_0(id(a)) = \delta_1(id(a)) = a$ for all $a \in O_G$; it is *with pairing* if its class O_G of objects forms a monoid; it is *monoidal* if it is reflexive with pairing and also its class of arrows forms a monoid, such that, if 0 is the neutral element of O_G, then $id(0)$ is the neutral element of A_G. □

We can think of a signature Σ as a graph, whose nodes are (underlined) natural numbers, and its arcs are univocally labeled by an operator, such that $f : \underline{n} \to \underline{1}$ iff $f \in \Sigma_n$. The usual notion of term can be formalized along this intuition, which allows to recover also alternative structures.

Definition 2.2 (graph theories). Given a signature Σ, the associated *graph theory* $G(\Sigma)$ is the monoidal graph with objects the elements of the commutative monoid $(\underline{\mathbb{N}}, \otimes, \underline{0})$ of underlined natural numbers (where $\underline{0}$ is the neutral object and the sum is defined as $\underline{n} \otimes \underline{m} = \underline{n + m}$); and arrows those generated by the following inference rules:

$$(generators)\ \frac{f : \underline{n} \to \underline{1} \in \Sigma}{f : \underline{n} \to \underline{1} \in G(\Sigma)} \qquad (sum)\ \frac{s : \underline{n} \to \underline{m}, t : \underline{n'} \to \underline{m'}}{s \otimes t : \underline{n} \otimes \underline{n'} \to \underline{m} \otimes \underline{m'}}$$

$$(identities)\ \frac{\underline{n} \in \underline{\mathbb{N}}}{id_{\underline{n}} : \underline{n} \to \underline{n}}$$

satisfying the *monoidality* axiom $id_{\underline{n} \otimes \underline{m}} = id_{\underline{n}} \otimes id_{\underline{m}}$ for all $\underline{n}, \underline{m} \in \underline{\mathbb{N}}$. □

Identities could be given just for $\underline{0}, \underline{1}$, using the monoidality axiom to define inductively the operator for all the objects, so obtaining a finitary presentation of the theories. The solution we chose is equivalent, yet easier to describe, and it is used for all the auxiliary operators introduced in the next definitions.

Definition 2.3 (monoidal theories). Given a signature Σ, the associated *monoidal theory* $\mathbf{M}(\Sigma)$ is the monoidal graph with objects the elements of the commutative monoid $(\underline{\mathbb{N}}, \otimes, \underline{0})$ of underlined natural numbers and arrows

those generated by the following inference rules:

$$(\textit{generators})\ \frac{f : \underline{n} \to \underline{1} \in \Sigma}{f : \underline{n} \to \underline{1} \in \mathbf{M}(\Sigma)} \qquad (\textit{sum})\ \frac{s : \underline{n} \to \underline{m}, t : \underline{n'} \to \underline{m'}}{s \otimes t : \underline{n} \otimes \underline{n'} \to \underline{m} \otimes \underline{m'}}$$

$$(\textit{identities})\ \frac{\underline{n} \in \mathbb{N}}{id_{\underline{n}} : \underline{n} \to \underline{n}} \qquad (\textit{composition})\ \frac{s : \underline{n} \to \underline{m}, t : \underline{m} \to \underline{k}}{s; t : \underline{n} \to \underline{k}}$$

Moreover, the composition operator ; is associative, and the monoid of arrows satisfies the *functoriality* axiom

$$(s \otimes t); (s' \otimes t') = (s; s') \otimes (t; t')$$

(whenever both sides are defined); the *identity* axiom $id_{\underline{n}}; s = s = s; id_{\underline{m}}$ for all $s : \underline{n} \to \underline{m}$; and the monoidality axiom $id_{\underline{n} \otimes \underline{m}} = id_{\underline{n}} \otimes id_{\underline{m}}$ for all $\underline{n}, \underline{m} \in \mathbb{N}$. □

Further enriching the auxiliary structure, we are finally able to present the more expressive kind of theories we deal with in our paper, *algebraic theories.*

Definition 2.4 (algebraic theories). Given a signature Σ, the associated *algebraic theory* $\mathbf{A}(\Sigma)$ is the monoidal graph with objects the elements of the commutative monoid $(\mathbb{N}, \otimes, \underline{0})$ of underlined natural numbers and arrows those generated by the following inference rules:

$$(\textit{generators})\ \frac{f : \underline{n} \to \underline{1} \in \Sigma}{f : \underline{n} \to \underline{1} \in \mathbf{S}(\Sigma)} \qquad (\textit{sum})\ \frac{s : \underline{n} \to \underline{m}, t : \underline{n'} \to \underline{m'}}{s \otimes t : \underline{n} \otimes \underline{n'} \to \underline{m} \otimes \underline{m'}}$$

$$(\textit{identities})\ \frac{\underline{n} \in \mathbb{N}}{id_{\underline{n}} : \underline{n} \to \underline{n}} \qquad (\textit{composition})\ \frac{s : \underline{n} \to \underline{m}, t : \underline{m} \to \underline{k}}{s; t : \underline{n} \to \underline{k}}$$

$$(\textit{duplicators})\ \frac{\underline{n} \in \mathbb{N}}{\nabla_{\underline{n}} : \underline{n} \to \underline{n} \otimes \underline{n}} \qquad (\textit{dischargers})\ \frac{\underline{n} \in \mathbb{N}}{!_{\underline{n}} : \underline{n} \to \underline{0}}$$

$$(\textit{permutation})\ \frac{\underline{n}, \underline{m} \in \mathbb{N}}{\rho_{\underline{n},\underline{m}} : \underline{n} \otimes \underline{m} \to \underline{m} \otimes \underline{n}}$$

Moreover, the composition operator ; is associative, and the monoid of arrows satisfies the functoriality axiom

$$(s \otimes t); (s' \otimes t') = (s; s') \otimes (t; t')$$

(whenever both sides are defined); the identity axiom $id_{\underline{n}}; s = s = s; id_{\underline{m}}$ for all $s : \underline{n} \to \underline{m}$; the monoidality axioms

$$id_{\underline{n} \otimes \underline{m}} = id_{\underline{n}} \otimes id_{\underline{m}} \qquad \rho_{\underline{n} \otimes \underline{m},\underline{p}} = (id_{\underline{n}} \otimes \rho_{\underline{m},\underline{p}}); (\rho_{\underline{n},\underline{p}} \otimes id_{\underline{m}})$$

$$!_{\underline{n} \otimes \underline{m}} = !_{\underline{n}} \otimes !_{\underline{m}} \qquad \nabla_{\underline{n} \otimes \underline{m}} = (\nabla_{\underline{n}} \otimes \nabla_{\underline{m}}); (id_{\underline{n}} \otimes \rho_{\underline{n},\underline{m}} \otimes id_{\underline{m}})$$

$$!_{\underline{0}} = \nabla_{\underline{0}} = \rho_{\underline{0},\underline{0}} = id_{\underline{0}} \qquad \rho_{\underline{0},\underline{n}} = \rho_{\underline{n},\underline{0}} = id_{\underline{n}}$$

for all $\underline{n}, \underline{m}, \underline{p} \in \mathbb{N}$; the *coherence* axioms

$$\nabla_{\underline{n}}; (id_{\underline{n}} \otimes \nabla_{\underline{n}}) = \nabla_{\underline{n}}; (\nabla_{\underline{n}} \otimes id_{\underline{n}}) \qquad \nabla_{\underline{n}}; \rho_{\underline{n},\underline{n}} = \nabla_{\underline{n}}$$

$$\nabla_{\underline{n}}; (id_{\underline{n}} \otimes !_{\underline{n}}) = id_{\underline{n}} \qquad \rho_{\underline{n},\underline{m}}; \rho_{\underline{m},\underline{n}} = id_{\underline{n}} \otimes id_{\underline{m}}$$

for all $\underline{n}, \underline{m} \in \mathbb{N}$; and the *naturality* axioms

$$(s \otimes t); \rho_{\underline{m},\underline{q}} = \rho_{\underline{n},\underline{p}}; (t \otimes s)$$

$$s; !_{\underline{m}} = !_{\underline{n}} \qquad s; \nabla_{\underline{m}} = \nabla_{\underline{n}}; (s \otimes s)$$

for all $s : \underline{n} \to \underline{m}, t : \underline{p} \to \underline{q}$. □

As for identities, also permutation and the other auxiliary operators could be inductively extended to all $\underline{n} \in \mathbb{N}$ starting from the basic cases, interpreting in a constructive way the monoidality axioms.

Let us consider the signature $\Sigma_{\sigma\epsilon} = \bigcup_{i=0}^{2} \Sigma_i$, where $\Sigma_0 = \{a, b\}$, $\Sigma_1 = \{f, g\}$ and $\Sigma_2 = \{h\}$ (that same signature is also used in the following sections). Some of the elements in $\mathbf{A}(\Sigma_{\sigma\epsilon})$ are a_Σ; $f_\Sigma : \underline{0} \to \underline{1}$, f_Σ; $g_\Sigma : \underline{1} \to \underline{1}$, a_Σ; $\nabla_{\underline{1}}$; $(f_\Sigma \otimes id_{\underline{1}})$; $h_\Sigma : \underline{0} \to \underline{1}$, intuitively corresponding to the terms $f(a), g(f(x))$ and $h(f(a), a)$, respectively, for a given variable x. In fact, a classical result we already anticipated proves that algebraic theories are equivalent to the ordinary construction (as it can be found e.g. in [2]) for term algebras.

Proposition 2.5 (algebraic theories and term algebras). *Let Σ be a signature. Then for all $\underline{n}, \underline{m} \in \mathbb{N}$ there exists a one-to-one correspondence between the set of arrows from $\underline{n}$ to $\underline{m}$ of $\mathbf{A}(\Sigma)$ and the m-tuples of elements of the term algebra –over a set of n variables– associated to Σ.* □

The previous result states that each arrow $t_\Sigma : \underline{n} \to \underline{1}$ identifies an element t of the term algebra over the set $\{x_1, \ldots, x_n\}$: an arrow $\underline{n} \to \underline{m}$ is an m-tuple of such elements, and arrow composition is term substitution. Note that this correspondence *requires* that ∇ and ! are natural: if this were not the case, we get *s-monoidal theories* [10,12]. In these more concrete structures, such elements as $a_\Sigma; \nabla_{\underline{1}}; h_\Sigma$ and $(a_\Sigma \otimes a_\Sigma); h_\Sigma$, that intuitively represent the same term $h(a, a)$, are different. In fact, in the PhD thesis [10] of the first author it is shown that a fundamental property of correspondence holds between s-monoidal theories and *term graphs* (as defined e.g. in the introductory chapter of [8]): each arrow $t_\Sigma : \underline{n} \to \underline{m}$ identifies a term graph t over Σ with a specified m-tuple of roots and a specified n-tuple of variables nodes, and arrow composition is graph replacement.

The incremental description of algebraic theories has received little attention in the literature (see [15,20]), despite the relevant fact that, differently from the usual categorical construction, all the elements of the class $\mathbf{A}(\Sigma)$ are inductively defined, making a much handier tool to deal with. In fact, the relevant point for our discussion is that, although their definitions are more involved than the classical, set-theoretical ones, algebraic theories allow for a characterization of terms which is far more general, and at the same time more concrete, than the one allowed by the usual formalization of the elements of a term algebra, separating in a better way the "Σ-structure" from the additional algebraic structure that the *meta-operators* used in the ordinary description (like *substitution*) implicitly enjoy. In this view, ! and ∇ represent respectively *garbage collection* and *sharing* (as discussed in [7,5]). As an example, let us consider the constant a: as a generator, the corresponding arrow is $a_\Sigma : \underline{0} \to \underline{1}$, while, when considered as an element of the term algebra over $\{x_1, x_2\}$, the associated arrow is $!_{\underline{2}}; a_\Sigma : \underline{2} \to \underline{1}$, where $!_{\underline{2}}$ intuitively corresponds to the *garbaging* of the two variables. Also the difference between $a_\Sigma; \nabla_{\underline{1}}; h_\Sigma$ and $(a_\Sigma \otimes a_\Sigma); h_\Sigma$ has a similar justification: in the first element, the a is shared;

in the latter, it is not. For our purposes, the difference between shared and unshared – discharged and undischarged – subterms does not play a relevant part, while instead s-monoidal theories hold a fundamental röle in [12], when dealing with *truly concurrent semantics* in the setting of *process algebras.*

2.2 Describing Systems

In this section we recall the basic formulation of our framework, inspired both from the *rewriting logic* approach by Meseguer [23] and the sos approach by Plotkin [28]. Intuitively, an *algebraic rewriting system* is just a set of rules, each of them carrying information (i.e., expressing some conditions) on the possible behaviours of the terms to which they can be applied.

Definition 2.6 (algebraic rewriting systems). An *algebraic rewriting system* (ARS) $\mathcal{R}$ is a tuple $\langle \Sigma_\sigma, \Sigma_\tau, N, R \rangle$, where $\Sigma_\sigma, \Sigma_\tau$ are signatures, N is a set of rule names and R is a function $R : N \to \mathbf{A}(\Sigma_\sigma) \times G(\Sigma_\tau) \times G(\Sigma_\tau) \times \mathbf{A}(\Sigma_\sigma)$ such that for all $d \in N$, if $R(d) = \langle l, a, b, r \rangle$, then $l : \underline{n} \to \underline{m}, r : \underline{p} \to \underline{q}$ iff $a : \underline{n} \to \underline{p}, b : \underline{m} \to \underline{q}$. We usually write $d : l \xrightarrow[b]{a} r$. □

A *context system* [22] is just a very simple ARS, where $R : N \to \Sigma_\sigma \times G(\Sigma_\tau) \times G(\Sigma_\tau) \times \Sigma_\sigma$, with the further restriction that $a : \underline{1} \to \underline{1}$ for all $a \in \Sigma_\tau$ (hence, for all $d \in N$, if $R(d) = \langle l, a, b, r \rangle$ then l, r have the same source and target). *Term rewriting systems* [17], instead, are given by a pair $\langle \Sigma, R \rangle$ where Σ is an ordinary signature, and R is a set of *rules*, i.e., of pairs $\langle l, r \rangle$ for l, r elements of the term algebra over Σ. Hence, thanks to Proposition 2.5, they are just a very particular case of algebraic rewriting systems, where Σ_τ is empty: in the following, we will refer to these systems as *horizontal rewriting systems* (also HRS's), and a rule will be simply denoted as $d : l \longrightarrow r$. In fact, an HRS is what is called an *unconditional rewriting theory* in [23]. It is actually less general, since in this paper we decided to consider rewriting systems built over signatures Σ instead that over equational theories (Σ, E), even if the extension of ARS's to deal with them is quite straightforward.

Let us consider the signatures $\Sigma_{\sigma\epsilon}$ (already introduced) and $\Sigma_{\tau\epsilon} = \Sigma_1$, where $\Sigma_1 = \{u, v, w\}$. Our running example will be the algebraic rewriting system $\mathcal{R}_e = \langle \Sigma_{\sigma\epsilon}, \Sigma_{\tau\epsilon}, N_e, R_e \rangle$, where the function R_e is described by

$$R_e = \{d : a \xrightarrow[u]{\iota} b, d_1 : f \xrightarrow[v]{u} g, d_2 : f \xrightarrow[w]{v} f, d_3 : h \xrightarrow[w]{u \otimes v} !_{\underline{1}} \otimes g\}.$$

(where ι is a shorthand for the identitiy arrow $id_{\underline{0}} \in \mathbf{M}(\Sigma_{\tau\epsilon})$). The intuitive meaning of the rules is: for d, the element a can be rewritten to b, producing an effect u; for d_1, f can be rewritten to g, producing an effect v, whenever there exists a suitable rewriting with an effect u. Or, in the ordinary term rewriting view: the term $f(x)$ is rewritten to $g(x)$, producing an effect v, but the rule can be applied only if the subterm associated to x is rewritten with an effect u; and so on for the other rules.

An ARS $\mathcal{R}$ can be considered as a logical theory, and any rewriting –using rules in $\mathcal{R}$– as a sequent entailed by the theory. An *algebraic sequent* is then a 5-tuple $\langle \alpha, s, a, b, t \rangle$, where $s \to t$ is a rewrite step, α is a *proof term* (encoding of the causes of the step), a and b are respectively the input and output conditions, the *actions* associated to the rewrite. We say that s *rewrites to* t *via* α (using a *trigger* a and producing an *effect* b) if we obtain the sequent $\alpha : s \xrightarrow[b]{a} t$ by finitely many applications of a set of *inference rules.*

Definition 2.7 (algebraic tile logic). Let $\mathcal{R} = \langle \Sigma_\sigma, \Sigma_\tau, N, R \rangle$ be an ARS. We say that $\mathcal{R}$ *entails* the class $\mathbf{R}$ of the *algebraic sequents* $\alpha : s \xrightarrow[b]{a} t$ obtained by finitely many applications of the following inference rules:

basic rules

$$(\textit{generators})\ \frac{d : s \xrightarrow[b]{a} t \in R}{d : s \xrightarrow[b]{a} t \in \mathbf{R}}$$

$$(\textit{h-refl})\ \frac{s : \underline{n} \to \underline{m} \in \mathbf{A}(\Sigma_\sigma)}{id_s : s \xrightarrow[id_{\underline{m}}]{id_{\underline{n}}} s \in \mathbf{R}} \qquad (\textit{v-refl})\ \frac{a : \underline{n} \to \underline{m} \in \mathbf{M}(\Sigma_\tau)}{id_a : id_{\underline{n}} \xrightarrow[a]{a} id_{\underline{m}} \in \mathbf{R}};$$

composition rules

$$(\textit{p-comp})\ \frac{\alpha : s \xrightarrow[b]{a} t, \alpha' : s' \xrightarrow[b']{a'} t' \in \mathbf{R}}{\alpha \otimes \alpha' : s \otimes s' \xrightarrow[b \otimes b']{a \otimes a'} t \otimes t' \in \mathbf{R}}$$

$$(\textit{h-comp})\ \frac{\alpha : s \xrightarrow[c]{a} t, \alpha' : s' \xrightarrow[b]{c} t' \in \mathbf{R}}{\alpha * \alpha' : s; s' \xrightarrow[b]{a} t; t' \in \mathbf{R}}$$

$$(\textit{v-comp})\ \frac{\alpha : s \xrightarrow[b]{a} u, \alpha' : u \xrightarrow[b']{a'} t \in \mathbf{R}}{\alpha \cdot \alpha' : s \xrightarrow[b;b']{a;a'} t \in \mathbf{R}};$$

auxiliary rules

$$(\textit{perm})\ \frac{a : \underline{n} \to \underline{m}, b : \underline{n'} \to \underline{m'} \in \mathbf{M}(\Sigma_\tau)}{\rho_{a,b} : \rho_{\underline{n},\underline{n'}} \xrightarrow[b \otimes a]{a \otimes b} \rho_{\underline{m},\underline{m'}} \in \mathbf{R}}$$

$$(\textit{dupl})\ \frac{a : \underline{n} \to \underline{m} \in \mathbf{M}(\Sigma_\tau)}{\nabla_a : \nabla_{\underline{n}} \xrightarrow[a \otimes a]{a} \nabla_{\underline{m}} \in \mathbf{R}} \qquad (\textit{disch})\ \frac{a : \underline{n} \to \underline{m} \in \mathbf{M}(\Sigma_\tau)}{!_a : !_{\underline{n}} \xrightarrow[id_{\underline{0}}]{a} !_{\underline{m}} \in \mathbf{R}}.$$

□

The different sets of rules are self-explaining. Basic rules provide the generators of the sequents, together with suitable identity arrows, whose intuitive meaning is that an element of $\mathbf{A}(\Sigma_\sigma)$ or $\mathbf{M}(\Sigma_\tau)$ can be rewritten to itself (showing no effect/using no trigger, so to say). Composition rules provide all the possible ways in which sequents can be composed, while auxiliary rules are the counterpart of the auxiliary operators for algebraic theories.

Let us consider the ARS $\mathcal{R}_e$ we previously defined. It entails the sequent

$$\dfrac{\dfrac{\overline{d : a \xrightarrow[u]{\iota} b} \quad \overline{d_1 : f \xrightarrow[v]{u} g}}{d * d_1 : a; f \xrightarrow[v]{\iota} b; g}\,(h\text{-}comp.) \qquad \overline{d_2 : f \xrightarrow[w]{v} f}}{(d * d_1) * d_2 : a; f; f \xrightarrow[w]{\iota} b; g; f}\,(h\text{-}comp.)$$

where ι is a shorthand for $id_{\underline{0}}$ and the entailment is described in a natural deduction style. It also entails the sequent

$$\dfrac{\dfrac{\overline{d : a \xrightarrow[u]{\iota} b}}{d : a \xrightarrow[u]{\iota} b} \qquad \dfrac{\overline{d : a \xrightarrow[u]{\iota} b} \quad \overline{d_1 : f \xrightarrow[v]{u} g}}{d * d_1 : a; f \xrightarrow[v]{\iota} b; g}}{d \otimes (d * d_1) : a \otimes (a; f) \xrightarrow[u \otimes v]{\iota} b \otimes (b; g)} \qquad \overline{d_3 : h \xrightarrow[w]{u \otimes v} !_{\underline{1}} \otimes g}}{(d \otimes (d * d_1)) * d_3 : (a \otimes (a; f)); h \xrightarrow[w]{\iota} b; g; g}$$

where the entailment is still described in a natural deduction style, but without using the rule names.

The class $\mathbf{R}$ is too concrete, in the sense that many sequents that intuitively should represent the same rewrite have a different representation. An equivalence over sequents can then be considered as a way to abstract away from implementation details, identifying *computationally* equivalent derivations.

Definition 2.8 (abstract algebraic sequents). Let $\mathcal{R} = \langle \Sigma_\sigma, \Sigma_\tau, N, R \rangle$ be an ARS. We say that it entails the class $\mathbf{R}_E$ of *abstract algebraic sequents*: equivalence classes of algebraic sequents entailed by $\mathcal{R}$ modulo the set E of axioms, which are intended to apply to the corresponding proof terms. The set E contains three *associativity* axioms, stating that all the composition operators are associative; the functoriality axioms

$$(\alpha \cdot \beta) * (\gamma \cdot \delta) = (\alpha * \gamma) \cdot (\beta * \delta) \qquad (\alpha \otimes \beta) * (\gamma \otimes \delta) = (\alpha * \gamma) \otimes (\beta * \delta)$$
$$(\alpha \otimes \beta) \cdot (\gamma \otimes \delta) = (\alpha \cdot \gamma) \otimes (\beta \cdot \delta)$$

(satisfied whenever both sides are defined); the identity axioms $id_s \cdot \alpha = \alpha = \alpha \cdot id_t$ and $id_b * \alpha = \alpha = \alpha * id_a$ for all $\alpha : s \xrightarrow[b]{a} t$; the monoidality axioms

$$id_{s \otimes t} = id_s \otimes id_t \qquad id_{a \otimes b} = id_a \otimes id_b$$
$$id_{s;t} = id_s * id_t \qquad id_{a;b} = id_a \cdot id_b$$
$$\alpha \otimes id_{id_{\underline{0}}} = \alpha = id_{id_{\underline{0}}} \otimes \alpha \qquad \rho_{a \otimes b, c} = (id_a \otimes \rho_{b,c}) * (\rho_{a,c} \otimes id_b)$$
$$!_{a \otimes b} = !_a \otimes !_b \qquad \nabla_{a \otimes b} = (\nabla_a \otimes \nabla_b) * (id_a \otimes \rho_{a,b} \otimes id_b)$$
$$!_{id_{\underline{0}}} = \nabla_{id_{\underline{0}}} = \rho_{id_{\underline{0}}, id_{\underline{0}}} = id_{id_{\underline{0}}} \qquad \rho_{id_{\underline{0}}, a} = \rho_{a, id_{\underline{0}}} = a$$

for all $\alpha \in \mathbf{R}$, $s, t \in \mathbf{A}(\Sigma_\sigma)$ and $a, b, c \in \mathbf{M}(\Sigma_\tau)$; the *coherence* axioms

$$\nabla_a * (id_a \otimes \nabla_a) = \nabla_a * (\nabla_a \otimes id_a) \qquad \nabla_a * \rho_{a,a} = \nabla_a$$
$$\nabla_a * (id_a \otimes !_a) = id_a \qquad \rho_{a,b} * \rho_{b,a} = id_a \otimes id_b$$

for all $a, b \in \mathbf{M}(\Sigma_\tau)$; and the *naturality* axioms

$$(\alpha \otimes \alpha') * \rho_{b,b'} = \rho_{a,a'} * (\alpha' \otimes \alpha)$$
$$\alpha * !_b = !_a \qquad \alpha * \nabla_b = \nabla_a * (\alpha \otimes \alpha)$$

for all $\alpha : s \xrightarrow[b]{a} t, \alpha' : s' \xrightarrow[b']{a'} t' \in \mathbf{R}$. □

This axiomatization properly extends the one given for unconditional rewriting logic in [23]. Note also that, as already happened for the theories of Section 2.1, even in this case we could have inductively defined identities, permutations and the other auxiliary operators starting from the basic cases, interpreting in a constructive way the monoidality axioms.

As an example, if we consider the ARS $\mathcal{R}_e$, from identity and monoidality axioms we have that

$$d \otimes (d * d_1) = (d \otimes d) * (id_u \otimes d_1)$$

hence the entailed proof terms

$$(d \otimes (d * d_1)) * d_3 \qquad ((d \otimes d) * (id_u \otimes d_1)) * d_3$$

are equivalent, even if the latter has a derivation unrelated to the one already shown for $(d \otimes (d * d_1)) * d_3$.

3 Operational Semantics for CCS

It is quite common in *concurrency theory* to deal with formalisms relying on the notion of *side-effects* and *synchronization* in determining the actual behaviour of a system, features wich are quite difficult to recast in frameworks like (classical) term rewriting. *Process (Description) Algebras* [3,14,24] offer a constructive way to describe *concurrent systems*, considered as structured entities (the *agents*) interacting by means of some synchronization mechanism. They define each system as a term of an algebra over a set of process constructors, building new systems from existing ones, on the assumption that algebraic operators represent basic features of a concurrent system. We present here one of the better known example of process algebra, the *Calculus of Communicating Systems* (CCS), introduced by Milner in the early Eighties [24], restricting ourselves, for the sake of exposition, to the case of *finite* CCS.

Definition 3.1 (the Calculus of Communicating Systems). Let *Act* be a set of atomic *actions*, ranged over by μ, with a distinguished symbol τ and equipped with an involutive function $\overline{\mu}$ such that $\overline{\tau} = \tau$. Moreover, let $\alpha, \overline{\alpha}, \ldots$ range over $Act \backslash \{\tau\}$. A CCS *process* (also *agent*) is a term generated by the following syntax

$$P ::= nil, \mu.P, P\backslash_\alpha, P[\Phi], P_1 + P_2, P_1 || P_2$$

where $\Phi : Act \to Act$ is a *relabeling* function, preserving involution and τ. Usually, we let $P, Q, R, \ldots$ range over the set *Proc* of processes. □

In the following, we indicate as Σ_{ccs} the signature associated with CCS processes (for example, *nil* is a constant, μ a unary operator for each element in *Act*, and so on...). Given a process P, its dynamic behaviour can be described by a suitable transition system, along the lines of the SOS approach, where the transition relation is freely generated from a set of inference rules.

Definition 3.2 (operational semantics of CCS. The ccs *transition system* is the relation $T_{ccs} \subseteq Proc \times Act \times Proc$ inductively generated from the following set of axioms and inference rules

$$\frac{}{\mu.P \xrightarrow{\mu} P} \text{ for } \mu \in Act \qquad \frac{P \xrightarrow{\mu} Q}{P[\Phi] \xrightarrow{\Phi(\mu)} Q[\Phi]} \text{ for } \Phi \text{ relabeling}$$

$$\frac{P \xrightarrow{\mu} Q}{P\backslash_\alpha \xrightarrow{\mu} Q\backslash_\alpha} \text{ for } \mu \notin \{\alpha, \overline{\alpha}\}$$

$$\frac{P \xrightarrow{\mu} Q}{P + R \xrightarrow{\mu} Q} \qquad \frac{P \xrightarrow{\mu} Q}{R + P \xrightarrow{\mu} Q}$$

$$\frac{P \xrightarrow{\mu} Q}{P||R \xrightarrow{\mu} Q||R} \qquad \frac{P \xrightarrow{\alpha} Q, P' \xrightarrow{\overline{\alpha}} Q'}{P||P' \xrightarrow{\tau} Q||Q'} \qquad \frac{P \xrightarrow{\mu} Q}{R||P \xrightarrow{\mu} R||Q}$$

where $P \xrightarrow{\mu} Q$ means that $\langle P, \mu, Q\rangle \in T_{ccs}$. □

A process P can execute an action μ and become Q if we can *inductively* construct a sequence of rule applications, such that the *transition* $\langle P, \mu, Q\rangle \in T_{ccs}$. As an example, to infer that from $P = (\alpha.nil + \beta.nil)||\overline{\alpha}.nil$ we can deduct $P \xrightarrow{\alpha} Q = nil||\overline{\alpha}.nil$, three different rules must be applied. Moreover, a process P can be rewritten into Q if there exists a *computation* from P to Q, i.e., a chain $P = P_0 \xrightarrow{\mu_1} P_1 \ldots P_{n-1} \xrightarrow{\mu_n} P_n = Q$ of one-step reductions.

The operational semantics we just defined is however too intensional, and more abstract semantics have been introduced by defining suitable *behavioural equivalences*, which identify processes exhibiting the same *observational behaviour*. Most of them are defined on the basic notion of *bisimulation* [27]: intuitively, two processes P, Q are *bisimilar* if, whenever P performs an action μ evolving to a state P', then also Q may execute that same action, evolving to a state Q' which is still bisimilar to P'.

Definition 3.3 (bisimulation equivalence). A symmetric equivalence relation $\sim_b \subseteq Proc \times Proc$ is a *bisimulation* if, whenever $P \sim_b Q$ for generic P, Q processes, then for any transition $P \xrightarrow{\mu} P'$ there exists a corresponding transition $Q \xrightarrow{\mu} Q'$ with $Q \sim_b Q'$. The maximal bisimulation equivalence is called *strong bisimulation*, and denoted by $\sim$. □

It is well-known that strong bisimilarity for ccs is also a congruence, and that it can be described by an equational theory over Σ_{ccs} [24]. In [3], the authors defined a *finitary* equational theory for an observational equivalence over their *Algebra of Communicating Processes*, introducing auxiliary operators. An obvious extension of their formalism can be adapted to get a finite description of strong bisimilarity for ccs, introducing three auxiliary operators, which intuitively split the parallel operator into three distinct cases, corresponding to left, right and synchronous composition of the sub-agents. On the other hand, Moller [25] has proved that bisimilarity cannot be finitely axiomatized without resorting to auxiliary operators. So, let Σ_{eccs} be the signature obtained extending Σ_{ccs} with the operators $\{\lfloor, \rfloor, | : 2 \to 1\}$.

Definition 3.4 (B-K axioms). Let P, Q be CCS processes. The *Bergstra-Klop* (also *B-K*) *axiomatization* is given by the following axioms for the parallel, relabelling and restriction operators

$$P||Q = ((P\lfloor Q) + (P \rfloor Q)) + (P|Q);$$

$$(\mu.P)\lfloor Q = P \rfloor (\mu.Q) = \mu.(P||Q);$$

$$(\mu.P)|(\mu'.Q) = \begin{cases} \tau.(P||Q) \text{ if } \mu' = \overline{\mu} \text{ and } \mu \neq \tau, \\ nil \qquad \text{otherwise;} \end{cases}$$

$$(\mu.P)\backslash_\alpha = \begin{cases} \mu.(P\backslash_\alpha) \text{ if } \mu \notin \{\alpha, \overline{\alpha}\}, \\ nil \qquad \text{otherwise;} \end{cases}$$

$$(\mu.P)[\Phi] = \Phi(\mu).(P[\Phi]);$$

$$nil\lfloor P = P \rfloor nil = nil|P = P|nil = nil\backslash_\alpha = nil[\Phi] = nil;$$

extended with the *Hennessy-Milner* (also *H-M*) axioms for the choice operator

$$P + P = P + nil = P \quad P + Q = Q + P \quad (P + Q) + R = P + (Q + R).$$

We usually write $P \sim_{BK} Q$ if P and Q are in the same equivalence class with respect to the B-K axioms. □

The H-M axioms simply state the associativity, commutativity, identity and idempotency of the non-deterministic operator (see [13]). The importance of the B-K axioms is given by their soundness and completness with respect to the bisimulation equivalence, as stated in the following result. ¿From our point of view, however, equally relevant is the fact that these axioms can be easily turned into rewriting rules, obtaining a confluent rewriting system, that identifies bisimilar agents: more on this in the next section.

Proposition 3.5 (B-K axioms and strong bisimulation). *Let P, Q be* CCS *processes. Then $P \sim Q$ iff $P \sim_{BK} Q$.* □

4 Operational Semantics from Rewriting Systems

In this section we show how the CCS operational semantics can be recovered by suitable rewriting systems. In particular, in Section 4.1 we define an algebraic rewriting system $\mathcal{R}_{ccs}$ which faithfully corresponds to the CCS transition system T_{ccs}. Then, in Section 4.2 we define the notion of *tile bisimulation*, roughly identifying sequents with the same effect: when applied to the sequents entailed by $\mathcal{R}_{ccs}$, it provides a recasting of strong bisimilarity for CCS processes. Finally, in Section 4.3 we describe a horizontal rewriting system $\mathcal{R}_{BK}$, that derives, for each element of a class of bisimilar CCS processes, a canonical representative of the class itself.

4.1 Using Tiles for CCS

As shown in the previous section, from an operational point of view a process algebra can be faithfully described by a triple $\langle \Sigma, A, R \rangle$, where Σ is the

signature of the algebra of agents, A is the set of actions and R is the set of deduction rules. Note that these rules are *conditional*: you need information on the *action* performed by the transitions in the premise, before applying a rule. Moreover, the rewriting steps are always performed *on top*: the order in which the rewrites are actually executed is important since, as an example, the correct operational behaviour of the agent $P = \alpha.\beta.nil$ is expressed saying that it executes first α and then β. If we let A_{ccs} be the signature containing all the atomic actions of Act (i.e., $A_{ccs} = \{\mu : \underline{1} \to \underline{1} \mid \mu \in Act\}$), then both those features are easily described in the framework of tile logic.

Definition 4.1 (the CCS rewriting system). The ARS $\mathcal{R}_{ccs}$ associated to CCS is the tuple $\langle \Sigma_{ccs}, A_{ccs}, N, R \rangle$, with the following set of rules:

$$act_\mu : \mu \xrightarrow[\mu]{id_{\underline{1}}} id_{\underline{1}} \qquad rel_\Phi : \Phi \xrightarrow[\Phi(\mu)]{\mu} \Phi$$

$$res_\alpha : \backslash_\alpha \xrightarrow{\mu} \backslash_\alpha \qquad \text{for} \quad \mu \notin \{\alpha, \overline{\alpha}\}$$

$$\langle + : + \xrightarrow[\mu]{\mu \otimes id_{\underline{1}}} id_{\underline{1}} \otimes !_{\underline{1}} \qquad + \rangle : + \xrightarrow[\mu]{id_{\underline{1}} \otimes \mu} !_{\underline{1}} \otimes id_{\underline{1}}$$

$$\xi_l : \| \xrightarrow[\mu]{\mu \otimes id_{\underline{1}}} \| \qquad \xi_r : \| \xrightarrow[\mu]{id_{\underline{1}} \otimes \mu} \| \qquad \xi_s : \| \xrightarrow{\alpha \otimes \overline{\alpha}} \|$$

(where we omitted the subscripts for the sake of readibility). □

Note that there is exactly one basic rule for each operational rule of CCS; some of them (such as act_μ and rel_Φ) are parametric with respect to the set of actions or to the set of relabeling functions, since the corresponding rules are so. The effect μ indicates that the process is actually "running", outputting the action μ. For example, the rule act_μ prefixes an idle process with the action μ, and then starts the execution, consuming that same action. There are also three rules dealing with the parallel operator: ξ_s synchronizes two running processes, while ξ_l and ξ_r perform an asynchronous move, taking a running and an idle process.

As an example of sequent construction, let us consider again the process $P = \alpha.\beta.nil$, executing sequentially first the action α, then the action β. The computation is represented by the sequent

$$(id_{nil;\beta} * act_\alpha) \cdot (id_{nil} * act_\beta) : nil; \beta; \alpha \xrightarrow[\alpha;\beta]{\iota} nil$$

whose two-steps entailment is the following

$$\frac{\overline{id_{nil;\beta} : nil; \beta \xrightarrow[\iota]{\iota} nil; \beta} \quad \overline{act_\alpha : \alpha \xrightarrow[\alpha]{\iota} id_{\underline{1}}}}{id_{nil;\beta} * act_\alpha : nil; \beta; \alpha \xrightarrow[\alpha]{\iota} nil; \beta}$$

$$\frac{\overline{id_{nil} : nil \xrightarrow[\iota]{\iota} nil} \quad \overline{act_\beta : \beta \xrightarrow[\beta]{\iota} id_{\underline{1}}}}{id_{nil} * act_\beta : nil; \beta \xrightarrow[\beta]{\iota} nil}$$

(where ι is a shorthand for both $id_{\underline{0}}$ and $id_{\underline{1}}$, since no confusion can arise), showing the importance of effects in expressing the ordering constraints: P can execute α only if the underlying process $P' = \beta.nil$ is actually idle.

For the agent $P = ((\alpha.nil)\backslash_\beta)\backslash_\gamma$, instead, the execution of the action α is represented by the sequent $((id_{nil} * act_\alpha) * res_\beta) * res_\gamma$, whose entailment is

$$\dfrac{\dfrac{\dfrac{\overline{id_{nil} : nil \xrightarrow[\iota]{} nil} \quad \overline{act_\alpha : \alpha \xrightarrow[\alpha]{} id_{\underline{1}}}}{id_{nil} * act_\alpha : nil; \alpha \xrightarrow[\alpha]{} nil} \quad \overline{res_\beta : \backslash_\beta \xrightarrow[\alpha]{} \backslash_\beta}}{(id_{nil} * act_\alpha) * res_\beta : nil; \alpha; \backslash_\beta \xrightarrow[\alpha]{} nil; \backslash_\beta} \quad \overline{res_\gamma : \backslash_\gamma \xrightarrow[\alpha]{} \backslash_\gamma}}{((id_{nil} * act_\alpha) * res_\beta) * res_\gamma : nil; \alpha; \backslash_\beta; \backslash_\gamma \xrightarrow[\alpha]{} nil; \backslash_\beta; \backslash_\gamma}$$

where the basic sequent act_α has been provided with a suitable context.

Note that the axioms impose an equivalence relation over sequents (i.e., over computations), and then offer a description that, even if more concrete than the one given by the set–theoretical relation entailed by a transition system, is still somewhat "abstract": there are many derivations that are identified, corresponding to "essentially" equivalent CCS computations. There is however an obvious adequacy result, stated by the following theorem.

Proposition 4.2 (computational correspondence). *Let P, Q be* CCS *agents, and P_R, Q_R the associated elements of $\mathbf{A}(\Sigma_{ccs})$. Then the transition $P \xrightarrow{\mu} Q$ is entailed by the* CCS *transition system T_{ccs} iff an abstract algebraic sequent $\alpha : P_R \xrightarrow[\mu]{id_{\underline{0}}} Q_R$ is entailed by $\mathbf{R}_{ccs}$.* □

The correspondence is instead one-to-one if we consider the restriction $\mathbf{R}_p$ of $\mathbf{R}_{ccs}$ over $\mathbf{A}(\Sigma_{ccs}) \times G(A_{ccs}) \times G(A_{ccs}) \times \mathbf{A}(\Sigma_{ccs})$: i.e., the relation obtained by dropping the proof terms from sequents. Or, equivalently, if we take into account the class of abstract sequents modulo the set of axioms E', obtained adding to E the axiom

$$\dfrac{\alpha : s \xrightarrow[b]{a} t, \beta : s \xrightarrow[b]{a} t}{\alpha = \beta}.$$

Proposition 4.3 (interleaving correspondence). *Let P, Q be* CCS *agents, and P_R, Q_R the associated elements of $\mathbf{A}(\Sigma_{ccs})$. Then the transition $P \xrightarrow{\mu} Q$ is entailed by the* CCS *transition system T_{ccs} (i.e., $\langle P, \mu, Q\rangle \in T_{ccs}$) iff the abstract algebraic sequent (modulo the set of axioms E') $\alpha : P_R \xrightarrow[\mu]{id_{\underline{0}}} Q_R$ is entailed by $\mathbf{R}_{ccs}$ (i.e., $\langle P_R, id_{\underline{0}}, \mu, Q_R\rangle \in \mathbf{R}_p$).* □

4.2 Recovering Bisimulation for Tiles

It seems quite reasonable that the notion of bisimulation could be extended to deal with our framework. In this section we introduce *tile bisimulation*, showing its (intuitive) correspondence with strong bisimilarity for CCS processes.

Definition 4.4 (tile bisimulation). Let $\mathcal{R} = \langle \Sigma_\sigma, \Sigma_\tau, N, R\rangle$ be an ARS. A symmetric equivalence relation $\equiv_b \subseteq \mathbf{A}(\Sigma_\sigma) \times \mathbf{A}(\Sigma_\sigma)$ is a *tile bisimulation* for $\mathcal{R}$ if, whenever $s \equiv_b t$ for generic s, t elements of $\mathbf{A}(\Sigma_\sigma)$, then for any abstract sequent $\alpha : s \xrightarrow[b]{a} s'$ entailed by $\mathcal{R}$ there exists a corresponding one $\beta : t \xrightarrow[b]{a} t'$ with $s' \equiv_b t'$. The maximal tile bisimulation equivalence is called *strong tile bisimulation*, and denoted by $\equiv_{st}$. □

This is an obvious generalization of Definition 3.3, due to the more concrete representation of states and the richer structure on effects shown by sequents with respect to ccs transitions. But of course there is a complete coincidence between bisimilarity over ccs processes and tile bisimilarity over the corresponding elements of $\mathbf{A}(\Sigma_{ccs})$.

Proposition 4.5 (bisimulation corrispondence). *Let P, Q be* ccs *agents, and P_R, Q_R the associated elements of $\mathbf{A}(\Sigma_{ccs})$. Then $P \sim Q$ iff $P_R \equiv_{st} Q_R$.* □

We need now to develop a concept analogous to congruence. Usually, an equivalence is a congruence whenever it preserves the operators. In our case, this "operator preserving" property can be restated in terms of parallel and horizontal composition.

Definition 4.6 (tile functoriality). Let $\mathcal{R} = \langle \Sigma_\sigma, \Sigma_\tau, N, R \rangle$ be an ARS. A symmetric equivalence relation $\equiv_f \subseteq \mathbf{A}(\Sigma_\sigma) \times \mathbf{A}(\Sigma_\sigma)$ is *functorial* for $\mathcal{R}$ if, whenever $s \equiv_f t, s' \equiv_f t'$ for generic s, s', t, t' elements of $\mathbf{A}(\Sigma_\sigma)$, then $s; s' \equiv_f t; t'$ (whenever defined) and $s \otimes s' \equiv_f t \otimes t'$. □

It is not in general true that a tile bisimulation equivalence is also functorial. The following results provide a characterization of such a property in terms of *tile decomposition.*

Definition 4.7 (tile decomposition). Let $\mathcal{R}$ be an ARS. We say that $\mathcal{R}$ verifies the *(tile) decomposition property* if *i)* whenever it entails an abstract sequent $\alpha : s; t \xrightarrow[b]{a} u$, then it entails also two sequents $\beta : s \xrightarrow[c]{a} s'$ and $\gamma : t \xrightarrow[b]{c} t'$ with $u = s'; t'$; and *ii)* whenever it entails an abstract sequent $\alpha : s \otimes t \xrightarrow[b]{a} u$, then it entails also two sequents $\beta : s \xrightarrow[b_1]{a_1} s'$ and $\gamma : t \xrightarrow[b_2]{a_2} t'$ with $u = s' \otimes t'$, $a = a_1 \otimes a_2$ and $b = b_1 \otimes b_2$. □

A very simple system not verifying the (tile) decomposition property is given by $\mathcal{R}_a = \langle \Sigma_{an}, \Sigma_a, N_a, R_a \rangle$, where $\Sigma_{an} = \{nil : 0 \to 1, a : 1 \to 1\}$, $\Sigma_a = \{a_1 : 1 \to 1, a_2 : 1 \to 1\}$ and

$$R_a = \{act : nil; a \xrightarrow[a_1]{id} nil, cons : a \xrightarrow[a_1]{a_1} id_{\underline{1}}\}.$$

The basic sequent *act* cannot be decomposed, while its source obviously can.

Proposition 4.8 (decomposition and bisimulation). *Let $\mathcal{R}$ be an* ARS. *If it verifies the decomposition property, then the associated strong tile bisimulation is functorial.* □

The converse is not true. In fact, the tile bisimulation associated to the ARS $\mathcal{R}_a$ is functorial, and it is freely generated from the basic classes $\{nil\}, \{id_{\underline{0}}\}, \{id_{\underline{1}}\}, \{a, a; a, \ldots\} = \{a^n | n \geq 1\}$, but the system does not verify the decomposition property. Note also the importance of $a_2 \in \Sigma_a$, which is responsible for the non-equivalence of $id_{\underline{1}}$ and a: on the contrary, that equivalence would have destroyed functoriality.

While it may be difficult to check out if a given rewriting system is "decomposable", the following proposition provides an easy syntactical property that implies decomposition.

Proposition 4.9 (basic source and decomposition). *Let $\mathcal{R}$ be the* ARS *$\langle \Sigma_\sigma, \Sigma_\tau, N, R\rangle$ such that, for all $d : s \xrightarrow[b]{a} t \in R$, $s \in \Sigma_\sigma$ (hence, the source is a basic operator). Then $\mathcal{R}$ satisfies the decomposition property.* □

Proof (sketch). The proof can be carried out in two steps.

First, each abstract sequent α can be decomposed into the vertical composition $\alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_n$ of "concrete" sequents α_i such that the operator $\cdot$ does not appear in any of them. These kind of sequents are called *one-step*, and they can be obtained without using the *v-comp* rule.

Then, let us suppose that $\alpha : s \xrightarrow[b]{a} t$ is one-step. Now, since the source of each rule must be a basic operator, we have that the structure of α exactly mirrors the one of its source s. And since also the axioms of algebraic sequents mirror those of algebraic theories, the result holds. □

In fact, both $\mathcal{R}_{ccs}$ and $\mathcal{R}_e$ verify this "basic source" property, hence the decomposition one, so that the following corollary holds.

Corollary 4.10 (strong bisimulation is functorial). *The strong tile bisimulation $\equiv_{st}$ associated to $\mathcal{R}_{ccs}$ is functorial.*

Thanks to Proposition 4.5, this result implies that strong bisimilarity for ccs processes is also a congruence. In fact, if an equivalence is functorial it preserves contexts, and *a fortiori* also operators. As an example, let P, Q be ccs agents, P_R, Q_R the associated elements of $\mathbf{A}(\Sigma_{ccs})$, and let us assume that $P \sim Q$. Hence $P_R \equiv_{st} Q_R$ (also $Q_R \equiv_{st} P_R$ by symmetry) and, by functoriality, $(P_R \otimes Q_R); || \equiv_{st} (Q_R \otimes P_R); ||$, so that also $P||Q \sim Q||P$ holds.

In general, it should be worthy to identify suitable "formats" for the rules such that, given a rewriting system $\mathcal{R}$, then whenever its rules fit a format then $\mathcal{R}$ is decomposable. An analogous work has been done on process algebras: see e.g. [1] for more details on the so-called GSOS format. For our tile model, some preliminary considerations can be found in [12].

4.3 B-K Axioms as Rewriting Rules

The aim of this section is to show that the axiomatization given in Definition 3.4 can be turned into a horizontal rewriting system, which is adequate for bisimilarity, in the sense that, given two ccs processes, they are bisimilar iff they may evolve to the same element.

Definition 4.11 (B-K axioms as rewriting rules). The HRS $\mathcal{R}_{BK}$ associated to the B-K axioms is the tuple $\langle \Sigma_{eccs}, \emptyset, N_{BK}, R_{BK}\rangle$, with the following set of rules:

$$dec : || \longrightarrow \nabla_{\underline{2}}; ((\nabla_{\underline{2}}; (\lfloor \otimes \rfloor); +) \otimes |); +$$

$$\sigma_l : (\mu \otimes id_{\underline{1}}); \lfloor \longrightarrow ||; \mu \quad \sigma_{le} : (nil \otimes id_{\underline{1}}); \lfloor \longrightarrow !_{id_{\underline{1}}}; nil$$

$$\sigma_r : (id_{\underline{1}} \otimes \mu); \rfloor \longrightarrow ||; \mu \quad \sigma_{re} : (id_{\underline{1}} \otimes nil); \rfloor \longrightarrow !_{id_{\underline{1}}}; nil$$

$$\sigma : (\alpha \otimes \overline{\alpha}); | \longrightarrow ||; \tau \quad \sigma_e : (\mu \otimes \mu'); | \longrightarrow !_{\underline{2}}; nil \text{ for } \mu' \neq \overline{\mu} \text{ or } \mu = \tau$$

$$\sigma_{nl} : (nil \otimes id_{\underline{1}}); | \longrightarrow !_{\underline{1}}; nil \quad \sigma_{nr} : (id_{\underline{1}} \otimes nil); | \longrightarrow !_{\underline{1}}; nil$$

$$res_\alpha : \mu; \backslash_\alpha \longrightarrow \backslash_\alpha; \mu \text{ for } \mu \notin \{\alpha, \overline{\alpha}\}$$

$$res_e : \alpha; \backslash_\alpha \longrightarrow !_{\underline{1}}; nil \quad res_n : nil; \backslash_\alpha \longrightarrow nil$$
$$rel_\Phi : \mu; \Phi \longrightarrow \Phi; \Phi(\mu) \quad rel_n : nil; \Phi \longrightarrow nil$$

together with the rules for for the choice operator

$$idem : \nabla_{\underline{1}}; + \longrightarrow id_{\underline{1}} \quad idnil : (id_{\underline{1}} \otimes nil); + \longrightarrow id_{\underline{1}}$$
$$comm : + \longrightarrow \rho_{\underline{1},\underline{1}}; + \quad assoc : (id_{\underline{1}} \otimes; +); + \longrightarrow (+ \otimes id_{\underline{1}}); +$$

(where we omitted the subscripts for the sake of readibility). With $\mathcal{R}_{BBK}$ we denote the HRS obtained without the rules for the choice operator; with $\mathcal{R}_I$ the one containing only the rules *idem* and *idnil* and finally with $\mathcal{R}_{AC}$ the one containing only the rules *comm* and *assoc*. □

The system we defined is convergetnt but not terminating: it is well-known that the axioms for associativity and commutativity of an operator cannot be in general turned into terminating rules. In fact, it is easy to see that

$$+ \longrightarrow \rho_{\underline{1},\underline{1}}; + \longrightarrow \rho_{\underline{1},\underline{1}}; \rho_{\underline{1},\underline{1}}; + = + \longrightarrow \rho_{\underline{1},\underline{1}}; + \longrightarrow \ldots$$

However, let $\Sigma_{sccs} = \{nil, \mu, +\} \subseteq \Sigma_{ccs}$ be the signature of *sequential* CCS (SCCS) processes: next result shows that $\mathcal{R}_{BK}$ is still adequate with respect to strong bisimulation.

Proposition 4.12 (bisimulation as normal form, I). *Let P, Q be* ECCS *processes. Then $P \sim_{BK} Q$ iff there exists a* SCCS *process S such that $\mathcal{R}_{BK}$ entails two sequents $\alpha : P \longrightarrow S$ and $\beta : Q \longrightarrow S$.* □

Notice that the normalization procedure is totally orthogonal to the usual notion of transition in the sos framework. In fact, let us consider the CCS processes $P = \alpha.\beta.nil$ and $Q = ((\alpha.nil)\backslash_\beta)\backslash_\gamma$: the associated computations evolving from them have been shown in the previous section. Note instead that, from a normalization point of view, P cannot move. Instead, Q *sequentially* executes two different *res* operations (one causally dependent from the other), and finally it evolves to $\alpha.nil$, as shown by the following sequents

$$\dfrac{\dfrac{\dfrac{\overline{id_{nil} : nil \longrightarrow nil} \quad \overline{res_\beta : \alpha; \backslash_\beta \longrightarrow \backslash_\beta; \alpha}}{id_{nil} * res_\beta : nil; \alpha; \backslash_\beta \longrightarrow nil; \backslash_\beta; \alpha} \quad \overline{id_{\backslash_\gamma} : \backslash_\gamma \longrightarrow \backslash_\gamma}}{(id_{nil} * res_\beta) * id_{\backslash_\gamma} : nil; \alpha; \backslash_\beta; \backslash_\gamma \longrightarrow nil; \backslash_\beta; \alpha; \backslash_\gamma}$$

$$\dfrac{\overline{id_{nil;\backslash_\beta} : nil; \backslash_\beta \longrightarrow nil; \backslash_\beta} \quad \overline{res : \alpha; \backslash_\gamma \longrightarrow \backslash_\gamma; \alpha}}{id_{nil;\backslash_\beta} * res_\gamma : nil; \backslash_\beta; \alpha; \backslash_\gamma \longrightarrow nil; \backslash_\beta; \backslash_\gamma; \alpha}$$

$$\dfrac{\overline{res_e : nil; \backslash_\beta \longrightarrow nil} \quad \overline{id_{\backslash_\gamma;\alpha} : \backslash_\gamma; \alpha \longrightarrow \backslash_\gamma; \alpha}}{res_e * id_{\backslash_\gamma;\alpha} : nil; \backslash_\beta; \backslash_\gamma; \alpha \longrightarrow nil; \backslash_\gamma; \alpha}$$

$$\dfrac{\overline{res_e : nil; \backslash_\gamma \longrightarrow nil} \quad \overline{id_\alpha : \alpha \longrightarrow \alpha}}{res_e * id_\alpha : nil; \backslash_\gamma; \alpha \longrightarrow nil; \alpha}$$

such that, by the axioms of Definition 2.8, we have

$$((id_{nil} * res_\beta) * id_{\backslash_\gamma}) \cdot (res_e * res_\gamma) \cdot (res_e * id_\alpha) : nil; \alpha; \backslash_\beta; \backslash_\gamma \longrightarrow nil; \alpha.$$

The above denotation of the abstract sequent in the example suggests a reduction where two steps are executed in parallel. Proposition 4.12 also has a stronger formulation, which is stated by the following result.

Proposition 4.13 (bisimulation as normal form, II). *Let P, Q be* ECCS *processes. Then $P \sim_{BK} Q$ iff there exists a* SCCS *process S such that the sequents $\alpha : P \longrightarrow_{BBK} P' \longrightarrow_{AC} P'' \longrightarrow_I S$ and $\beta : Q \longrightarrow_{BBK} Q' \longrightarrow_{AC} Q'' \longrightarrow_I S$ are entailed by $\mathcal{R}_{BK}$ (where $P \longrightarrow_{BBK} P'$ means that it is entailed by $\mathcal{R}_{BBK}$, and so on).* □

Since both $\mathcal{R}_{BBK}$ and $\mathcal{R}_I$ are terminating, then, if we considered an equational extension of our tile model, each class of bisimilar processes would be provided with a normal form, modulo associativity and commutativity.

References

[1] L. Aceto, B. Bloom, F.W. Vaandrager, *Turning SOS Rules into Equations*, Information and Computation **111** (1), pp. 1-52.

[2] J.A. Goguen, J.W. Tatcher, E.G. Wagner, J.R. Wright, *Initial Algebra Semantics and Continuous Algebras*, Journal of the ACM **24** (1), 1977, pp. 68-95.

[3] J.A. Bergstra, J.W. Klop, *Process Algebra for Synchronous Communication*, Information and Computation **60**, 1984, pp. 109-137.

[4] A. Corradini, F. Gadducci, *CPO Models for Infinite Term Rewriting*, in Proc. AMAST'95, LNCS 936, 1995, pp. 368–384.

[5] A. Corradini, F. Gadducci, U. Montanari, *Relating Two Categorical Models of Concurrency*, in Proc. RTA'95, LNCS 914, 1995, pp. 225–240.

[6] A. Corradini, U. Montanari, *An Algebraic Semantics for Structured Transition Systems and its Application to Logic Programs*, Theoretical Computer Science **103**, 1992, pp. 51-106.

[7] A. Corradini, *Term Rewriting, in Parallel*, submitted.

[8] M.C.J.D. van Eekelen, M.J. Plasmeijer, M.R. Sleep, *Term Graph Rewriting, Theory and Practice*, John Wiley & Sons, 1993.

[9] G. Ferrari, U. Montanari, *Towards the Unification of Models for Concurrency*, in Proc. CAAP'90, LNCS 431, 1990, pp. 162-176.

[10] F. Gadducci, *On the Algebraic Approach to Concurrent Term Rewriting*, PhD Thesis, Università di Pisa, Pisa. Technical Report TD-96-02, Department of Computer Science, University of Pisa, 1996.

[11] F. Gadducci, U. Montanari, *Enriched Categories as Models of Computations*, in *Proc. Fifth Italian Conference on Theoretical Computer Science, ICTCS'95*, World Scientific, 1996, pp. 1-24.

[12] F. Gadducci, U. Montanari, *The Tile Model*, Technical Report TR-96-27, Department of Computer Science, University of Pisa, 1996.

[13] M. Hennessy, R. Milner, *Algebraic Laws for Nondeterminism and Concurrency*, Journal of the ACM **32** (1), 1985, pp. 137-161.

[14] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[15] B. Jacobs, *Semantics of Weakening and Contraction*, Annals of Pure and Applied Logic **69**, 1994, pp 73-106.

[16] R. Keller, *Formal Verifications of Parallel Programs*, Communications of ACM **7**, 1976, pp. 371-384.

[17] J.W. Klop, *Term Rewriting Systems*, in Handbook of Logic in Computer Science, Vol. I, eds. S. Abramsky, D. Gabbay, and T. Maibaum, Oxford University Press, 1992, pp. 1-116.

[18] A. Kock, G.E. Reyes, *Doctrines in Categorical Logic*, in Handbook of Mathematical Logic, ed. John Bairwise, North Holland, 1977, pp. 283-313.

[19] G.M. Kelly, R.H. Street, *Review of the Elements of 2-categories*, Lecture Notes in Mathematics 420, 1974, pp. 75-103.

[20] J. Lafont, *Equational Reasoning with 2-dimensional Diagrams* in Term Rewriting, French Spring School of Theoretical Computer Science, Font-Romeu, May 1993, LNCS 909, 1995, pp. 170-195.

[21] F. W. Lawvere, *Functorial Semantics of Algebraic Theories*, Proc. National Academy of Science **50**, 1963, pp. 869-872.

[22] K.G. Larsen, L. Xinxin, *Compositionality Through an Operational Semantics of Contexts*, in Proc. ICALP'90, LNCS 443, 1990, pp. 526-539.

[23] J. Meseguer, *Conditional Rewriting Logic as a Unified Model of Concurrency*, Theoretical Computer Science **96**, 1992, pp. 73-155.

[24] R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[25] F. Moller, *The Non-Existence of Finite Axiomatizations for CCS Congruences*, in Proc. LICS'90, IEEE Press, 1990, pp. 142-153.

[26] N. Martí-Oliet, J. Meseguer, *Rewriting Logic as a Logical and Semantic Framework*, SRI Technical Report, CSL-93-05, 1993.

[27] D. Park, *Concurrency and Automata on Infinite Sequences*, in Proc. Fifth G-I Conference, LNCS 104, 1981, pp. 167-183.

[28] G. Plotkin, *A Structural Approach to Operational Semantics*, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[29] A.J. Power, *An Abstract Formulation for Rewrite Systems*, in Proc. CTCS'89, LNCS 389, 1989, pp. 300-312.

[30] W. Reisig, *Petri Nets*, Springer-Verlag, 1985.

Electronic Notes in Theoretical Computer Science 4 (1996)

Modelling Conditional Rewriting Logic in Structured Categories

Hiroyuki Miyoshi [1]

The College of Crosscultural Communication and Business
Shukutoku University
1150-1 Fujikubo, Miyoshi-machi, Iruma-gun, Saitama 354, Japan
miyoshi@ccb.shukutoku.ac.jp

Abstract

We reformulate and generalize the functorial model of Meseguer's *conditional* full rewriting logic by using inserter, a weighted limit in 2-categories. Indeed 2-categories are categories enriched in **Cat**. Therefore this method also can be extended to sesqui-categories and other enriched categories, with which we can model various aspects of rewritings and strategies.

1 Introduction

J. Meseguer introduced his conditional rewriting logic in [12,13], and gave a functorial semantics for it in [13] and a 2-algebraic theory semantics in [12]. But his treatments of conditionals are not fully 2-categorical because he essentially uses subequalizers. As Lambek pointed out in his original paper [11], a subequalizer is not a 2-categorical limit in the sense that this does not require the 2-dimensional universality.

In 2-category (or enriched category) theory, the *weighted limit* was proposed as the more suitable notion than that of 2-limit [7,1]. Our first approach is to use a weighted limit in 2-categories, *inserter*, instead of subequalizer. By using inserters, we define 2-categorical models of Meseguer's conditional rewriting logic. In 2-category **Cat**, this limit coincides with subequalizer because of the completeness of **Cat** for weighted limits [7]. In this case, our model is the same that Meseguer defined in [12]. In other words, Meseguer's functorial model is a nontrivial instance of our 2-categorical models.

The weighted limit was originally defined in enriched categories, and a 2-category is a category enriched in **Cat**, that is, a **Cat**-category. Our formulation has one advantage in that **Cat** can be replaced with other suitable monoidal categories. So we replace **Cat** with $\mathbf{Cat}'$ and construct a model of

[1] This work is partially supported by Shukutoku University.

© *1996 Elsevier Science B. V.*

the conditional flat rewriting logic in $\mathbf{Cat}'$-categories (a model of unconditional logic in the $\mathbf{Cat}'$-category $\mathbf{Cat}'$ is described in [3]).

Next, from the previous two experiences, we propose to model rewriting with variaous enriched categories. which are called (enriched) categorical rewriting models. As leading examples, we provide three monoidal categories **SGray**, **Gray** and $\mathbf{Gray}^{op}$, and consider enrichment in them. All these monoidal categories are the category of 2-categories **2Cat** with various monoidal structures. Roughly speaking, a **SGray**-categorical model is a sesqui-categorical models with concurrency presented by 2-cells in hom-2-categories; **Gray**- and $\mathbf{Gray}^{op}$-categorical models can be considered as representing inner-first and outer-first rewriting strategies.

This paper is constituted as follows. In section 2, we discuss our model of Meseguer's rewriting logic. this is based on 2-categories and inserters. In section 3, as preliminaries of following sections, we briefly introduce enriched categories and related definitions. The content of previous section can be stated with the terms of enriched category theory, because a 2-category is an category enriched in **Cat**. In section 4, we introduce sesqui-categories, i.e. $\mathbf{Cat}'$-categories, and apply the previous method to sesqui-categories, constructing the models of nonoverlapping conditional rewritings. In section 5, we propose $\mathcal{V}$-categorical rewriting models, and pick up three examples. Finally, in section 6, we state conclusions and some further directions.

This paper contains various concepts of category theory. Their mathematics is not so hard but sometimes long. Following the workshop comittee's advice, we state only essential definitions and ideas; many proofs are omitted.

2 Models of conditional full rewriting logic

2.1 weighted limit in 2-categories

Here we assume the ordinary category theory and basic notions about 2-category. In this section, we define the notion of the weighted limit in 2-categories. The standerd references to 2-categories and their weighted limits are [9] and [8]. But note that this is are a special case of the general definition in enriched categories (see section 3).

Definition 2.1 (weighted limit in 2-categories). Let $\mathcal{B}$ be a 2-category, $\mathcal{G}$ a small 2-category. Given two 2-functors $F : \mathcal{G} \to \mathbf{Cat}$ (*weight*) and $G : \mathcal{G} \to \mathcal{B}$ (*diagram*), we can construct a 2-functor $[\mathcal{G}, \mathbf{Cat}](F, \mathcal{B}(?, G-))$ sending B to $[\mathcal{G}, \mathbf{Cat}](F, \mathcal{B}(B, G-))$. If this 2-functor has a representation

$$\mathcal{B}(B, \{F, G\}) \cong [\mathcal{G}, \mathbf{Cat}](F, \mathcal{B}(B, G-)) \tag{1}$$

with unit

$$\mu : F \to \mathcal{B}(\{F, G\}, G-),$$

the pair $(\{F, G\}, \mu)$, or simply an object $\{F, G\}$ in $\mathcal{B}$ is called a *F-weighted limit* of G. □

If F is a 2-functor to $\mathbf{1}$ (the terminal object of **Cat**), then we can obtain

ordinary 2-limits. These 2-limits are called *conical limit* in the context of weighted limits.

The one-dimensional aspect of the representation,

$$\mathcal{B}_0(B, \{F, G\}) \cong [\mathcal{G}, \mathbf{Cat}]_0(F, \mathcal{B}(B, G-)) \tag{2}$$

does not, in general, suffice to exhibit $(\{F, G\}, \mu)$ as the limit (for example, sections 3.7 and 3.8 of [7]). It does, howerer, suffice to determine $(\{F, G\}, \mu)$ uniquely within isomorphism, since the **Set**-valued functor on the right of (2) is also represented by $(\{F, G\}, \mu)$. ¿From this, we can see that the subequaliser coincides with an weighted limit (inserter) in the 2-category **Cat**.

Definition 2.2 Let $\mathcal{G}$ be

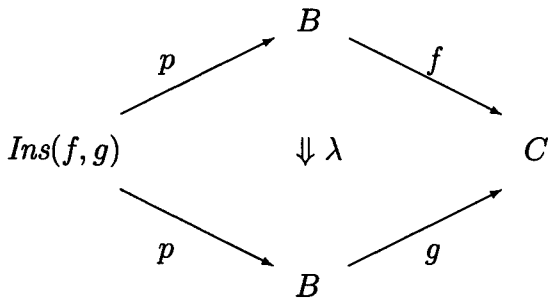
$$\cdot \rightrightarrows \cdot$$

and the images of $\mathcal{G}$ by F and G respectively

$$\mathbf{1} \underset{1}{\overset{0}{\rightrightarrows}} \mathbf{2} \qquad B \underset{g}{\overset{f}{\rightrightarrows}} C$$

Then we call the weighted limit made of these data the *inserter* of f and g, and denote $\{F, G\}$ as $Ins(f, g)$. □

Resolving by the definition, the inserter is the following diagram



which satisfies universality conditions. That is, the inserter of f and g is presented by the triple $(Ins(f, g), p, \lambda)$.

Proposition 2.3 *the subequalizer coincides with an inserter in* **Cat**.

Proof. Straightforward from the completeness of **Cat** as a ordinary category. For the general case, see Remark 3.9. □

2.2 Meseguer's conditional full rewriting logic

A *(labelled) rewrite theory* $\mathcal{R} = (\Sigma, E, L, R)$ consists of an algebra $A = (\Sigma, E)$, a set of *labels* L, and a set of *rewrite rules* R, which is $R \subset L \times (T_{\Sigma,E}(X)^2)^+$, a set of pairs consisting of an unique label and a non-empty sequence of *sequents* that are pairs of E-equivalence classes of Σ-term on a countable infinite set of variables X. A sequent $([t], [t'])$ is denoted in $[t] \to [t']$, and A rewrite rule $(r, ([t], [t']), ([u_1], [v_1]), \ldots, ([u_k], [v_k]))$ is written as

$$r : [t] \to [t'] \text{ if } [u_1] \to [v_1] \wedge \cdots \wedge [u_k] \to [v_k]$$

$[u_1] \to [v_1] \wedge \cdots \wedge [u_k] \to [v_k]$ is called the *condition* of this rule.

A *conditional full rewriting logic* is an inference system of sequents depending on a rewrite theory. We show its inference rules as follows; at the same time we present the construction of *proof terms* $\gamma : [t] \to [t']$.

(i) Reflexivity: For each $[t] \in T_{\Sigma,E}(X)$,

$$\frac{}{[t] : [t] \to [t]}.$$

(ii) Congruence: For each $f \in \Sigma_n$,

$$\frac{\alpha_1 : [t_1] \to [t'_1] \ \ldots \ \alpha_n : [t_n] \to [t'_n]}{f(\alpha_1, \ldots, \alpha_n) : [f(t_1, \ldots, t_n)] \to [f(t'_1, \ldots, t'_n)]}.$$

(iii) Full Replacement: For each rewrite rule

$$r : [t(\overline{x}^n)] \to [t'(\overline{x}^n)] \text{ if } [u_1(\overline{x}^n)] \to [v_1(\overline{x}^n)] \wedge \cdots \wedge [u_k(\overline{x}^n)] \to [v_k(\overline{x}^n)]$$

in R,

$$\frac{\alpha_1 : [w_1] \to [w'_1] \ldots \alpha_n : [w_n] \to [w'_n] \quad \beta_1 : [u_1(\overline{w}/\overline{x})] \to [v_1(\overline{w}/\overline{x})] \ldots \beta_k : [u_k(\overline{w}/\overline{x})] \to [v_k(\overline{w}/\overline{x})]}{r(\overline{\alpha}^n, \overline{\beta}^k) : [t(\overline{w}/\overline{x})] \to [t'(\overline{w'}/\overline{x})]}.$$

(iv) Transitivity: For arbitrary $[t_1], [t_2], [t_3] \in T_{\Sigma,E}(X)$,

$$\frac{\alpha : [t_1] \to [t_2] \qquad \beta : [t_2] \to [t_3]}{\beta \circ \alpha : [t_1] \to [t_3]}.$$

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sequent $[t] \to [t']$ and write

$$\mathcal{R} \vdash [t] \to [t']$$

iff $[t] \to [t']$ is obtained by finite application of the above rules.

2.3 $\mathcal{R}$-model

In this section we define the *$\mathcal{R}$-model* as a model of conditional rewriting logic. This is a similar concept to models of algebraic theories or sketches in the 1-category case. Here we adopt a relatively elementary setting.

Definition 2.4 Given a rewriting logic $\mathcal{R} = (\Sigma, E, R, L)$, a *$\mathcal{R}$-model* $\mathcal{M}$ is a pair of a 2-categry $\mathcal{C}$ and an interpretation $(-)_\mathcal{M}$ satisfying the following conditions:

(i) $\mathcal{C}$ has 2-products, that is, conical limits on any finite discrete category $\mathcal{G}$, and products in base category are 2-products. In paticular, there is 2-initial object 1, the limit on the empty category.

(ii) The base category has a (Σ, E)-algebra structure. That is, assigning to a sort a fixed object, an arrow an operation, and defining the composition of operations as that of arrows, the diagrams corresponding to equations in $A = (\Sigma, E)$ commute (this interpretation is denoted as $(-)_\mathcal{M}$). For an equivalence class $[t]$ of a term t, this interpretation is defined as follows:
(a) To each sort, assign an object S.

(b) To each operation symbol $f \in \Sigma_n$, assign an arrow $f_{\mathcal{M}} : S^n \to S$ in $\mathcal{C}$, where S^n is the n product of S.

(c) Fix a canonical order in the countable set of variables. When the variables occurring in the Σ-term t is included in a finite sequence of variables $\theta = (x_1, \ldots, x_k)$ in this canonical order, define $t^{\theta}_{\mathcal{M}}$ inductively:

t is an variable $x_i (1 \leq i \leq k)$:

$$x_{i\mathcal{M}}^{\theta} = \pi_i : S^k \to S,$$

where π_i is the i-projection of the product S^k.

$t = f(t_1, \ldots, t_n)$:

$$t^{\theta}_{\mathcal{M}} = f_{\mathcal{M}} \circ \langle t_{i\mathcal{M}}^{\theta}, \ldots, t_{n\mathcal{M}}^{\theta} \rangle : S^k \to S,$$

where $\langle t_{i\mathcal{M}}^{\theta}, \ldots, t_{n\mathcal{M}}^{\theta} \rangle$ is the product of arrows, that is, an arrow detemined by universality of the product.

Here for an equation $t = t'$ in E, letting θ be the sequence of variables occuring in both sides, $t^{\theta}_{\mathcal{M}} = t'^{\theta}_{\mathcal{M}}$.

Hereafter, if θ is the minimum sequence determined by the context, we omit θ and write $t^{\theta}_{\mathcal{M}}$ as $t_{\mathcal{M}}$.

(iii) For each rewrite rule

$$r : [t] \to [t'] \text{ if } [u_1] \to [v_1] \wedge \cdots \wedge [u_k] \to [v_k]$$

in R, let arrows determined from $u_{i\mathcal{M}}, v_{i\mathcal{M}}$ by the universality of 2-product be

$$u_{\mathcal{M}} = \langle u_{1\mathcal{M}}, \ldots, u_{k\mathcal{M}} \rangle$$
$$v_{\mathcal{M}} = \langle v_{1\mathcal{M}}, \ldots, v_{k\mathcal{M}} \rangle,$$

then there is an inserter $(Ins(u_{\mathcal{M}}, v_{\mathcal{M}}), p_{\mathcal{M}}, \lambda_{\mathcal{M}})$ of $u_{\mathcal{M}}, v_{\mathcal{M}}$, and so is a 2-cell

$$r_{\mathcal{M}} : t_{\mathcal{M}} \circ p_{\mathcal{M}} \Rightarrow t'_{\mathcal{M}} \circ p_{\mathcal{M}}.$$

□

Definition 2.5 A $\mathcal{R}$-model $\mathcal{M}$ *satisfies* $[t] \to [t']$ iff there is an 2-cell $t_{\mathcal{M}} \Rightarrow t'_{\mathcal{M}}$. We denote this satisfaction relation as

$$\mathcal{M} \models [t] \to [t'].$$

When every $\mathcal{R}$-models satisfy $[t] \to [t']$, we write

$$\mathcal{R} \models [t] \to [t'].$$

□

2.4 Soundness and completeness

Theorem 2.6 *(Soundness). For a rewriting theory $\mathcal{R}$,*

$$\mathcal{R} \vdash [t] \to [t']$$

implies

$$\mathcal{R} \models [t] \to [t']$$

Proof(Outline) Essentially parallel to the proof in [13]. For every $\mathcal{R}$-model, we have to show that for each proof term

$$\gamma : [t] \to [t']$$

there is an 2-cell

$$\gamma_{\mathcal{M}} : t_{\mathcal{M}} \Rightarrow t'_{\mathcal{M}}.$$

By induction on the depth of proof terms, we can define $\gamma_{\mathcal{M}}$ only with constructions allowed in $\mathcal{R}$-models. The rest is to show the coincidence of dom^v and cod^v of $\gamma_{\mathcal{M}}$ with the interpretations of $[t]$ and $[t']$ given in $\mathcal{R}$-models; this is easily implied from the constructions of them. □

For completeness, we need to show that the term model $\mathcal{T}_{\mathcal{R}}(X)$ defined in [13] is an $\mathcal{R}$-model.

Proposition 2.7 *$\mathcal{T}_{\mathcal{R}}(X)$ is a $\mathcal{R}$-system.*

Proof. cf. [13]. □

Proposition 2.8 *A $\mathcal{R}$-system is a $\mathcal{R}$-model in* **Cat**. *Especially $\mathcal{T}_{\mathcal{R}}(X)$ is also an $\mathcal{R}$-model in* **Cat**.

Proof. We put a 2-category **Cat** on $\mathcal{C}$ in the definition of $\mathcal{R}$-models, then show that $\mathcal{R}$-systems satisfy the conditions 1–3 of that definition. The condition 1 is satisfied because **Cat** is complete for weighted limits. The condition 2 is evident by substituting $\mathcal{S}$ in S. For the condition 3, we can use the subqualizer triple as an inserter by Proposition 2.3. □

Theorem 2.9 *(Completeness). For a rewrite theory $\mathcal{R}$,*

$$\mathcal{R} \models [t] \to [t']$$

implies

$$\mathcal{R} \vdash [t] \to [t']$$

Proof. ¿From $\mathcal{R} \models [t] \to [t']$ and the previous proposition, we can obtain

$$\mathcal{T}_{\mathcal{R}}(\{x_1, \ldots, x_n\}) \models [t(x_1, \ldots, x_n)] \to [t'(x_1, \ldots, x_n)]$$

and there is a natural transformation

$$\alpha : [t(x_1, \ldots, x_n)] \to [t'(x_1, \ldots, x_n)] :$$

$$\mathcal{T}_{\mathcal{R}}(\{x_1, \ldots, x_n\})^n \to \mathcal{T}_{\mathcal{R}}(\{x_1, \ldots, x_n\})$$

which is a 2-cell in **Cat**. The component of this natural transformation at an object $[x_1] \times \ldots \times [x_n]$ is

$$\alpha([x_1], \ldots, [x_n]) : [t(x_1, \ldots, x_n)] \to [t'(x_1, \ldots, x_n)]$$

which is an equivalence class of proof term that is an arrow in $\mathcal{T}_{\mathcal{R}}(\{x_1, \ldots, x_n\})$. Therefore the sequent $[t(x_1, \ldots, x_n)] \to [t'(x_1, \ldots, x_n)]$ is provable in $\mathcal{R}$. □

3 Enriched category and weighted limit

In the previous section, our arguement has proceeded within the 2-category theory. But indeed weighted limits are defined in the larger framework of

enriched category theory. As a preparation for the following sections, we present the general definitions of enriched category and weighted limit in it.

The contents of this section are all extracted from [7] or [1]. Because of the limit of space, we omit pictorial diagrams and detailed proofs. Please consult with these references to them.

Definition 3.1 (monoidal category) A *monoidal category* $\mathcal{V} = (\underline{\mathcal{V}}, \otimes, I, a, l, r)$ consists of the following data:

(i) (underlying category) a category $\underline{\mathcal{V}}$

(ii) (monoidal product) a bifunctor $\otimes : \underline{\mathcal{V}} \times \underline{\mathcal{V}} \to \underline{\mathcal{V}}$

(iii) (unit) an object I in $\underline{\mathcal{V}}$

(iv) (associativity law) a natural isomorphism a whose components are $a_{X,Y,Z} : (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z)$ for each triple of objects X, Y, Z in $\underline{\mathcal{V}}$

(v) (left unit law) a natural isomorphism l whose components are $l_X : I \otimes X \to X$ for each objects X in $\underline{\mathcal{V}}$

(vi) (right unit law) a natural isomorphism r whose components are $r_X : X \otimes I \to X$ for each objects X in $\underline{\mathcal{V}}$

satisfying two coherence axioms:

$$a_{W,X,Y\otimes Z} \circ a_{W\otimes X,Y,Z} = 1 \otimes a_{X,Y,Z} \circ a_{W,X\otimes Y,Z} \circ a_{W,X,Y} \otimes 1$$

$$1 \otimes l_Y \circ a_{X,I,Y} = r_X \otimes 1$$

Definition 3.2 The monoidal category $\mathcal{V}$ is said to be *symmetric* if it has a natural isomorphism c whose components are $s_{X,Y} : X \otimes Y \to Y \otimes X$, satisfying the coherence axioms:

$$1 \otimes s_{X,Z} \circ a_{Y,X,Z} \circ x_{X,Y} \otimes 1 = a_{Y,Z,X} \circ s_{X,Y\times Z} \circ a_{X,Y,Z}$$

$$l_X \circ s_{X,I} = r_X$$

$$s_{Y,X} \circ s_{X,Y} = id_{X\otimes Y}$$

Definition 3.3 The monoidal category $\mathcal{V}$ is said to be *closed* if each functor $- \otimes Y : \underline{\mathcal{V}} \to \underline{\mathcal{V}}$ has a right adjoint $[Y, -]$. When both $- \otimes Y$ and $X \otimes -$ have right adjoints $[Y, -]_r$ and $[X, -]_l$ respectively, $\mathcal{V}$ is said to be *biclosed*. If $\mathcal{V}$ is symmetric and closed, then it is biclosed and $[Y, -] = [Y, -]_r = [X, -]_l$.

Definition 3.4 ($\mathcal{V}$-category) Let $\mathcal{V}$ be a monoidal category. A *$\mathcal{V}$-category* $\mathcal{A}$ consists of the following data:

(i) (objects) a class $Ob\mathcal{A}$

(ii) (hom-objects) an object $\mathcal{A}(A, B)$ in $\underline{\mathcal{V}}$ for each pair $A, B \in Ob\mathcal{A}$

(iii) (composition law) an arrow $c_{A,B,C} : \mathcal{A}(B, C) \otimes \mathcal{A}(A, B) \to \mathcal{A}(A, C)$ in $\underline{\mathcal{V}}$ for each triple $A, B, C \in Ob\mathcal{A}$

(iv) (identity element) an arrow $j_A : I \to \mathcal{A}(A, A)$ in $\underline{\mathcal{V}}$ for each object $A \in Ob\mathcal{A}$

satisfying the associativity axiom and the unit axiom:

$$c_{A,C,D} \circ c_{A,B,C} \otimes 1 = c_{A,B,D} \circ 1 \otimes c_{B,C,D} \circ a_{\mathcal{A}(A,B),\mathcal{A}(B,C),\mathcal{A}(C,D)}$$

$$l_{\mathcal{A}(A,B)} = c_{A,A,B} \circ j_A \otimes 1$$

$$r_{\mathcal{A}(A,B)} = c_{A,B,B} \circ 1 \otimes j_A$$

Example 3.5 Every locally small 2-category is an **Cat**-category.

Definition 3.6 ($\mathcal{V}$-functor) Given $\mathcal{V}$-categories $\mathcal{A}, \mathcal{B}$, a *$\mathcal{V}$-functor* $F : \mathcal{A} \to \mathcal{B}$ consists in giving:

(i) an object $F(A)$ in $Ob\mathcal{B}$ for every object $A \in Ob\mathcal{A}$

(ii) an arrow $F_{AB} : \mathcal{A}(A, B) \to \mathcal{B}(F(A), F(B))$ in $\underline{\mathcal{V}}$ for each pair $A, B \in Ob\mathcal{A}$

satisfying the composition axiom and the unit axiom:

$$F_{A,C} \circ c_{A,B,C} = c_{FA,FB,FC} \circ F_{A,B} \otimes F_{B,C}$$

$$F_{A,A} \circ j_A = j_{FA}$$

Definition 3.7 ($\mathcal{V}$-natural transformation) Let $\mathcal{A}, \mathcal{B}$ be $\mathcal{V}$-categories and $F, G : \mathcal{A} \to \mathcal{B}$ $\mathcal{V}$-functors. A *$\mathcal{V}$-natural transformation* $\alpha : F \Rightarrow G$ consists in giving an arrow $\alpha_A : I \to \mathcal{B}(F(A), G(A))$ in $\underline{\mathcal{V}}$ for every object $A \in Ob\mathcal{A}$, satisfying the $\mathcal{V}$-natural condition:

$$c_{FA,FB,GB} \circ F_{A,B} \otimes \alpha_B \circ r^{-1}_{\mathcal{A}(A,B)} = c_{FA,GA,GB} \circ \alpha_A \otimes F_{A,B} \circ l^{-1}_{\mathcal{A}(A,B)}$$

Henceforth we suppose $\mathcal{V}$ is symmetric monoidal closed category whose underlying ordinary category $\underline{\mathcal{V}}$ is locally small and complete, in the sense that it admits all small limits.

Definition 3.8 (weighted limit in $\mathcal{V}$-category). Let $\mathcal{B}$ be a $\mathcal{V}$-category, $\mathcal{G}$ a $\mathcal{V}$-category. Given two $\mathcal{V}$-functors $F : \mathcal{G} \to \mathcal{V}$ (weight) and $G : \mathcal{G} \to \mathcal{B}$ (diagram), we can construct a $\mathcal{V}$-functor $[\mathcal{G}, \mathcal{V}](F, \mathcal{B}(?, G-))$ sending B to $[\mathcal{G}, \mathcal{V}](F, \mathcal{B}(B, G-))$. If this $\mathcal{V}$-functor has a representation

$$\mathcal{B}(B, \{F, G\}) \cong [\mathcal{G}, \mathcal{V}](F, \mathcal{B}(B, G-)) \tag{3}$$

with unit

$$\mu : F \to \mathcal{B}(\{F, G\}, G-),$$

the pair $(\{F, G\}, \mu)$, or simply an object $\{F, G\}$ in $\mathcal{B}$ is called a *F-weighted limit of G*.

Remark 3.9 Applying $\underline{\mathcal{V}}(I, -) : \underline{\mathcal{V}} \to \mathbf{Set}$ to (3), we have a bijection of sets

$$\mathcal{B}_0(B, \{F, G\}) \cong [\mathcal{G}, \mathcal{V}]_0(F, \mathcal{B}(B, G-)) \tag{4}$$

In general, this condition for $(\{F, G\}, \mu)$ is strictly weaker than (3) and does not suffice to make $(\{F, G\}, \mu)$ the limit. However it suffices to *detect* $(\{F, G\}, \mu)$ as limit if the existance of the limit is known [7].

The weighted limit $\{F, G\}$ is *small* when the weight $F : \mathcal{G} \to \mathcal{V}$ is small, that is, $\mathcal{G}$ is small. We say the $\mathcal{V}$-category $\mathcal{B}$ to be *complete* if it admits all small weighted limits;

Proposition 3.10 *Suppose $\mathcal{V}$ is a symmetric monoidal closed category such that $\underline{\mathcal{V}}$ is locally small and also complete. Then $\mathcal{V}$-category $\mathcal{V}$ is complete.*

Proof. See [7], p.74. □

Example 3.11 The **Cat**-category **Cat** is complete.

4 Models of conditional flat rewriting logic

4.1 **Cat'** *and sesqui-category*

The category **Cat** has one symmetric monoidal closed structure, that is, the Cartesian closed structure. However it is not so well known that **Cat** has another symmetric monoidal closed structure as follows:

- Let $\mathcal{C}$ and $\mathcal{D}$ be small categories. The monoidal product $\mathcal{C} \otimes \mathcal{D}$ is defined to be the category such that:
  - an object is a pair $(X, Y) \in Ob\mathcal{C} \times Ob\mathcal{D}$;
  - an arrow from (X, Y) to (X', Y') is a finite sequence of nonidentity arrows of $\mathcal{C}$ or $\mathcal{D}$, with one sequence of alternate arrows forming a directed path in $\mathcal{C}$, and the other forming a directed path in $\mathcal{D}$;
  - an identity is the empty sequence;
  - Composition is given by concatenation and cancellation by the composition of $\mathcal{C}$ or $\mathcal{D}$.
- A exponential object $[\mathcal{C}, \mathcal{D}]$ is a category whose objects are functors from $\mathcal{C}$ to $\mathcal{D}$ and, for each functor $F, G : \mathcal{C} \to \mathcal{D}$, whose arrows from F to G are maps which assign to each object $X \in Ob\mathcal{C}$ an arrow from $F(X)$ to $G(X)$; these maps are called *(unnatural) transformations.* Roughly speaking, this is a natural transformation without naturality.

This monoidal product can be defined by universality: a monoidal product $\mathcal{C} \otimes \mathcal{D}$ is the most universal category $\mathcal{A}$ such that $L_X : \mathcal{D} \to \mathcal{A}$ exists for each X in $Ob\mathcal{C}$, $R_Y : \mathcal{C} \to \mathcal{A}$ exists for each Y in $Ob\mathcal{D}$ and they satisfy $L_X Y = R_Y X$. The unit of this monoidal product is **1**.

Proposition 4.1 *The category* **Cat** *has* exactly *two symmetric monoidal closed structures [4]. The one is the Cartesian closed structure and another is that defined above.* **Cat** *with the above symmetric monoidal closed structure is denoted as* **Cat'**

Remark 4.2 Note that in **Cat'**, $\mathcal{C} \otimes \mathcal{D}$ is generated so that, for each $f : X \to X'$ in $\mathcal{C}$ and $g : Y \to Y'$ in $\mathcal{D}$, the following diagram

$$\begin{array}{ccc} (X,Y) & \xrightarrow{(f, id_Y)} & (X',Y) \\ {\scriptstyle (id_X, g)}\downarrow & & \downarrow{\scriptstyle (id_{X'}, g)} \\ (X,Y') & \xrightarrow[(f, id_{Y'})]{} & (X',Y') \end{array}$$

does not commute (we see (f, id_Y) as f, etc.); however, in **Cat** where the monoidal product is Cartesian product, it does.

Definition 4.3 A *sesqui-category* is a $\mathbf{Cat}'$-category.

Sesqui-categories were used as such a model of (unconditional) term rewriting that respects the length of derivation [16], and this idea was applied to flat (unconditional) rewriting logic [3].

As known from Remark 4.2, the key feature of the sesqui-categorical model of rewriting is that, when arrows $\alpha : u \Rightarrow u'$ in $\mathcal{A}(A, B)$ and $\beta : v \Rightarrow v'$ in $\mathcal{A}(B, C)$ are given, we have two arrows from $v \circ u$ to $v' \circ u'$ in $\mathcal{A}(A, C)$, one representing "α then β" and another "β then α." This means that this model respects the order of rewriting of nested terms.

To model conditional logics, we need such limit that works as "inserter" in sesqui-categorical models. Indeed, we can define a weighted limit in $\mathbf{Cat}'$, *sesqui-inserter*, analogus to the inserter in a 2-category, as follows:

Definition 4.4 $\mathcal{G}, F, G$ are the same as inserter. If the functor $[\mathcal{G}, \mathbf{Cat}'](F, \mathcal{B}(?, G-))$ has a representation

$$\mathcal{B}(B, \{F, G\}) \cong [\mathcal{G}, \mathbf{Cat}'](F, \mathcal{B}(B, G-)) \tag{5}$$

with unit

$$\mu : F \to \mathcal{B}(\{F, G\}, G-)$$

then the pair $(\{F, G\}, \mu)$ is the *sesqui-inserter* of F and G.

4.2 Conditional flat rewriting logic

A sesqui-category with some sesqui-inserters models the conditional *flat* rewriting logic whose inference rules are Reflexivity, Congruence, Transitivity and Flat Replacement, where

Flat Replacement: For each rewrite rule

$$r : [t(\overline{x}^n)] \to [t'(\overline{x}^n)] \text{ if } [u_1(\overline{x}^n)] \to [v_1(\overline{x}^n)] \wedge \cdots \wedge [u_k(\overline{x}^n)] \to [v_k(\overline{x}^n)]$$

in R,

$$\frac{[w_1] : [w_1] \to [w_1] \ \ldots \ [w_n] : [w_n] \to [w_n] \qquad \beta_1 : [u_1(\overline{w}/\overline{x})] \to [v_1(\overline{w}/\overline{x})] \ \ldots \ \beta_k : [u_k(\overline{w}/\overline{x})] \to [v_k(\overline{w}/\overline{x})]}{r(\overline{[w]}^n, \overline{\beta}^k) : [t(\overline{w}/\overline{x})] \to [t'(\overline{w}/\overline{x})]}.$$

Given a rewrite theory $\mathcal{R}$, the entailment of $[t] \to [t']$ in the conditional flat rewriting logic is written as:

$$\mathcal{R} \vdash_{\text{flat}} [t] \to [t']$$

4.3 $\mathbf{Cat}'$-$\mathcal{R}$-model

The definition of $\mathbf{Cat}'$-$\mathcal{R}$-*model* can be obtained from that of $\mathcal{R}$-model by replacing inserters with sesqui-inserters and 2-products with $\mathbf{Cat}'$-product,

where $\mathbf{Cat}'$-product is defined as a weighted limit whose diagram is a $\mathbf{Cat}'$-functor from a discrete category and whose weight is a $\mathbf{Cat}'$-functor to $\mathbf{1}$, the terminal sesqui-category.

Because of completeness of $\mathbf{Cat}$ as an ordinary category, all $\mathbf{Cat}'$-categories are complete. Therefore the $\mathbf{Cat}'$-$\mathcal{R}$-model is defined for arbitrary conditional $\mathcal{R}$. Proof terms of conditional flat rewriting logic make a nontrivial term model by the same construction as $\mathcal{R}$-model.

The satification of $[t] \to [t']$ by every $\mathbf{Cat}'$-$\mathcal{R}$-models is written as:

$$\mathcal{R} \models_{\mathbf{Cat}'} [t] \to [t']$$

Theorem 4.5 *(Soundness and completeness). For a rewriting theory $\mathcal{R}$,*

$$\mathcal{R} \vdash_{\text{flat}} [t] \to [t']$$

if and only if

$$\mathcal{R} \models_{\mathbf{Cat}'} [t] \to [t']$$

Proof. Almost parallel to the 2-categorical case and [13]. □

5 Enriched categorical rewriting models

As Corradini et al. pointed out in [3], in 2-categorical and sesqui-categorical models of rewriting logics, categorical structures represent not only "denotational" semantics but also "operational" computations. Therefore, apart from symbolic logics, we can see those categorical structures itself as computational models. (But this does not mean that we abandon logic; in fact, algebraic theories used in [3] and [12] are examples of "categorical" logic).

Two models we have defined in previous sections are the special enriched categories which have some properties. Here, we say *($\mathcal{V}$-)categorical rewriting models* for similar categorical models of rewriting in $\mathcal{V}$-categories where $\mathcal{V}$ is the monoidal category of a categorical structure.

As concrete examples, we think about models in **SGray**-, **Gray**- and $\mathbf{Gray}^{op}$-categories. Like **Cat** and $\mathbf{Cat}'$, **SGray** is a **SGray**-category. Also **Gray** is a **Gray**-category and $\mathbf{Gray}^{op}$ is a $\mathbf{Gray}^{op}$-category. So these rewriting models have nontrivial instances.

SGray has a symmetric monoidal closed structure. therefore, for **SGray**-categories, we can define a inserter-like limit in the same fashion, and by which we can model conditional rewriting in **SGray**-categories. On the other hand, **Gray** and $\mathbf{Gray}^{op}$ are not symmetric, but they are biclosed. By the general results of [5], we can define weighted limits in biclosed case, but more subtle treatments are needed.

In this section, we mention only the ideas briefly, and concentrate to explain the role of 2-cells of hom-objects (hom-2-categories). Full definitions of each models will appear in [14].

5.1 **SGray**-*categorical rewriting models*

SGray, **Gray** and $\mathbf{Gray}^{op}$ are the category of small 2-categories **2Cat** with different monoidal structures (all these structures are defined in J. Gray's books [6] under the quite different naming convention).

The monoidal product of **SGray**, called the *strong Gray product*, is such that if $\mathcal{C}$ and $\mathcal{D}$ are small 2-categories, 2-cells of $\mathcal{C} \otimes \mathcal{D}$ are generated by 2-cells of $\mathcal{C}$ and $\mathcal{D}$, and invertible 2-cells (called *structure 2-cells*)

$$\begin{array}{ccc} (X,Y) & \xrightarrow{(f,id_Y)} & (X',Y) \\ {\scriptstyle (id_X,g)}\downarrow & \Downarrow & \downarrow{\scriptstyle (id_{X'},g)} \\ (X,Y') & \xrightarrow[(f,id_{Y'})]{} & (X',Y') \end{array}$$

for any $f : X \to X'$ in $\mathcal{C}$ and $g : Y \to Y'$ in $\mathcal{D}$, which subject to appropriate axioms. Symmetry of this product is obvious.

A exponential object $[\mathcal{C},\mathcal{D}]$ is a 2-category whose objects are 2-functors from $\mathcal{C}$ to $\mathcal{D}$; for each 2-functor $F, G : \mathcal{C} \to \mathcal{D}$, whose arrows α from F to G are maps which assign to each object $X \in Ob\mathcal{C}$ an arrow α_X from $F(X)$ to $G(X)$ such that, for any arrow $f : X \to Y$ of $\mathcal{C}$, the diagram

$$\begin{array}{ccc} F(X) & \xrightarrow{\alpha_X} & G(X) \\ {\scriptstyle F(f)}\downarrow & \Downarrow \alpha_f & \downarrow{\scriptstyle G(f)} \\ F(Y) & \xrightarrow[\alpha_Y]{} & G(Y) \end{array}$$

has an invertible 2-cell α_f of $\mathcal{D}$ (these maps are called *psude-natural transformations*); and whose 2-cells are modifications (here we don't explain them; see [6] and [1]).

By the strong Gray product, **SGray** is made to be a symmetric monoidal closed category, and **SGray** is a **SGray**-category. Comparing with Remark 4.2, a **SGray**-category can be regarded as a $\mathbf{Cat}'$-category with invertible structure 2-cells in hom-2-category.

Note that on the one hand, if, for every objects (2-categories) their monoidal products of **SGray**, we unify two parallel arrows which have an invertible 2-cell between them, and collapsing 2-categories to ordinary categories, then we have the same structure as the monoidal category **Cat**; on the other hand, if, for those things of **SGray**, we forget 2-cells and think of only base categories, then we have the same structure as the monoidal category $\mathbf{Cat}'$.

This means that, as categorical rewriting models, a **SGray**-categorical model can be seen as a $\mathbf{Cat}'$-categorical model with the information of concurrency in a **Cat**-categorical model. The information which 2-cells in hom-

2-category carry, is that about concurrency; if there is a 2-cell between 1-cells (paths), these data denote a concurrent computation.

5.2 **Gray**- *and* **Gray**^{op}*-categorical rewriting models*

The product of **Gray**, called the *Gray product*, is such that if $\mathcal{C}$ and $\mathcal{D}$ to be small 2-categories, 2-cells of $\mathcal{C} \otimes \mathcal{D}$ are generated by 2-cells of $\mathcal{C}$ and $\mathcal{D}$, and downward structure 2-cells

$$\begin{array}{ccc} (X,Y) & \xrightarrow{(f,id_Y)} & (X',Y) \\ {\scriptstyle (id_X,g)}\downarrow & \Downarrow & \downarrow{\scriptstyle (id_{X'},g)} \\ (X,Y') & \xrightarrow[(f,id_{Y'})]{} & (X',Y') \end{array}$$

for any $f : X \to X'$ in $\mathcal{C}$ and $g : Y \to Y'$ in $\mathcal{D}$, which subject to appropriate axioms. Comparing with Remark 4.2, **SGray** can be regarded as **Cat**$'$ with downward structure 2-cells in hom-2-category. The product of **Gray**^{op} is similar to that of **Gray** except that structure 2-cells are opposite.

As a categorical rewriting model, we can regard 2-cells of a hom-2-category as the preference of rewriting paths in that model. Therefore, we can see that **Gray**-rewriting models represent the *inner-first* rewriting strategy, and **Gray**^{op}-rewriting models represent the *outer-first* one. In fact, if we consider only the order structure of path, then models in a **LocOrd**$_l$-category [10] are suffice to represent the inner-first strategy.

6 Conclusion

What I have done is separated into two parts. Firstly, we gereralize functorial semantics and algebraic theory semantics, and construct enriched categorical models for the four rewriting logics (○ is made in this paper):

Rewriting Logic	Functorial	Alg. Theory	Enriched Cat.
uncond. full	[13]	[3]	○
cond. full	[13]*	[12]*	○
uncond. flat	→	[3]	○
cond. flat	→	→	○

(∗ discussed only 1-dimensional universality.)

Especially, conditional rewritings can be properly treated using weighted limits.

Secondary, we have discussed that our models in enriched categories, called $\mathcal{V}$-categorical rewriting models, are useful to model various aspects of rewritings other than the four rewriting logics above. As examples, we present the

three categorical rewriting models and explain what computation they model.

To finish this paper, we will list up the author's ongoing approaches. He hopes that some of them will appear in [14]:

(i) formulation of conditional rewriting logics for **SGray**-, **Gray**- and $\mathbf{Gray}^{op}$-categorical rewriting model;

(ii) application to rewriting of infinitary terms:
The author conjectures that **Cat**- and **Cat'**-categories internal to **CPO** present (conditional and unconditional) categorical rewriting models for infinitary term in the similar fashion. For a restricted case, it is treated in [2].

(iii) modelling more flexible concurrency controls and rewriting strategies:
By **Gray**- and $\mathbf{Gray}^{op}$-categories, we have modelled inner-most and outer-most rewriting strategies. But we want to describe concurrency controls and strategies more freely. To construct such categorical rewriting models, we might need a uniform principle to add $\mathcal{V}$ to arbitrary 2-cells as concurrency information, and some form of free construction;

(iv) combination with higher-dimensional computational model:
SGray-categorical rewriting models resemble in Pratt's paradigm of "modelling computation with geometry" [15]. But its descriptive power is restricted, because it does not have higher-order cells. For example, Our approach cannot express the situation in which pairs of computations (f, g), (g, h), (h, f) can proceed concurrently but the triple (f, g, h) cannot do. To merge both approaches fully, we might need a monoidal products of higher-dimensional categorical structures (complexes);

7 Acknowledgement

The 2-categorical models in this paper originally appeared in my talk at SLACS '94 Workshop in Japan, but has not been published. Through a recent conversation with Dr. Yoshiki Kinoshita at ETL, The author noticed that this approach can be applied to the sesqui-category case. Thus he decided to revise and present them. He is grateful to his critism and encouragement.

References

[1] Borceux, F.: *Handbook of Categorical Algebra 2*, Cambridge University Press, 1994.

[2] Corradini, A. and Gadducci, F.: CPO Models for Infinite Term Rewriting, *Algebraic Methodology and Software Technology* (Alagar, V. S. and Nivat M. (eds.)), LNCS 936, Springer, 1995, pp.368–384.

[3] Corradini, A., Gadducci, F. and Montanari, U.: Relating Two Categorical Models of Term Rewriting, *Rewiting Techniques and Application* (Hsiang, J. (ed)), LNCS 914, 1995.

[4] Foltz, F., Lair, C. and Kelly, G. M.: Algebraic Categories with Few monoidal Biclosed Structures or None, *Journal of Pure and Applied Algebra*, Vol. 17 (1980), pp. 171–177.

[5] Gordon, R. and Power, A. J.: Gabriel-Ulmer Duality for Categories Enriched in Bicategories, manuscript, submitted.

[6] Gray, J. W.: *Formal Category Theory: Adjointness for 2-categories*, Lecture Notes in Mathematics, Vol. 391, Springer-Verlag, 1974.

[7] Kelly, G. M.: *Basic Concepts of Enriched Category Theory*, London Mathematical Society Lecture Note Series, Vol. 64, Cambridge University Press, 1982.

[8] Kelly, G. M.: Elementary Observations on 2-Categorical Limits, *Bull. Austral. Math. Soc.*, Vol. 39 (1989), pp. 301–317.

[9] Kelly, G. M. and Street, R.: Review of the Elements of 2-categories, *Category Seminar* (Kelly, G. M. (ed)), Lecture Notes in Mathematics, Vol. 420, Springer-Verlag, 1974, pp. 75–103.

[10] Kinoshita, Y. and Power, J.: Lax Naturality through Enrichment, to appear in *Journal of Pure and Applied Algebra*.

[11] Lambek, J.: Subequalizers, *Canadian Mathematical Bulletin*, Vol. 13, No. 3 (1970), pp. 337–349.

[12] Meseguer, J.: *Rewriting as a Unified Model of Concurrency*, Technical Report, SRI-CSL-90-02R, SRI International, Computer Science Laboratory, 1990. Revised version.

[13] Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency, *Theoretical Computer Science*, Vol. 96 (1992), pp. 73–155.

[14] Miyoshi, H.: *Enriched Categorical Rewriting Models*, thesis in preparation.

[15] Pratt, V.: Modelling Concurrency with Geometry, *Proc. of the 18th ACM Symposium on Principles of Programming Language*, ACM Press, 1991, pp. 311-322.

[16] Stell, J. G.: Modelling Term Rewriting Systems by Sesqui-Categories, *Proc. Catégories, Algèbres, Esquisses et Néo-Esquisses*, 27-30 Sept. 1994.

Electronic Notes in Theoretical Computer Science 4 (1996)

ELAN: A logical framework based on computational systems

Peter Borovanský Claude Kirchner Hélène Kirchner
Pierre-Etienne Moreau Marian Vittek

INRIA Lorraine & CRIN-CNRS
615, rue du Jardin Botanique, BP 101
54602 Villers-lès-Nancy Cedex, France
`http://www.loria.fr/equipe/protheo.html`

Abstract

ELAN implements computational systems, a concept that combines rewriting logic with the powerful description of rewriting strategies. ELAN can be used either as a logical framework or to describe and execute deterministic as well as non-deterministic rule based processes. We present the general features of the language and outline some of the applications it has been used for.

1 Introduction

> **elan** n. **1.** Enthusiastic vigor and liveness. **2.** Style; flair. [Fr < OFr. *eslan*, rush < eslancer, to throw out: es-, out (< Lat. ex-) + *lancer*, to throw (< LLat. *lanceare*, to throw a lance < Lat. *lancea*, lance).]
> The American Heritage Dictionary

Starting from the idea that inference systems can be quite conveniently described by rewrite rules, we began in the early nineties the design and implementation of a language in which inference systems can be represented in a natural way, and executed reasonably efficiently. This led us quickly to formalise such a language using the conditional rewriting logic introduced by J. Meseguer [Mes92] and to see ELAN as a logical framework where the frame logic is rewriting logic [Vit94].

In ELAN, a logic can be expressed by specifying its syntax and its inference rules. The syntax of the logic can be described using mixfix operators as in the OBJ or Maude languages [GKK+87,CELM96]. The inference rules of the logic are described by conditional rewrite rules. In order to make the description executable, we introduced the notion of strategy [KKV95a]. A computational system consists of a rewriting theory plus a strategy description. The underlying concepts are thus extremely simple and a natural question was to understand how far one can go in this direction, and how usable such a framework is. From the implementation of an interpretor first and recently of a

© *1996 Elsevier Science B. V.*

compiler, we have experimented with the system many examples from small classical ones to large and complex ones. To summarize, ELAN provides the following main features:

- A semantics based on many-sorted rewriting logic,
- A powerful language to express strategies of rewrite rule application, including dont-care and dont-know choices on strategies,
- A general pre-processor making easier the translation of a logic into rewriting logic,
- A standard library to facilitate user developments,
- Modular constructions via local or global importations as well as parametric modules,
- A generic mixfix and user-definable syntax,
- Associative commutative (AC) operators in interpreted mode,
- A very efficient compiler for ELAN programs without AC operators.

The goal of this paper is to give a general presentation of the system and of some of the realisations it has been used for, in order to convince the reader that the approach is not only realistic but also extremely useful as a logical framework allowing to conduct both computations and deductions in a combined and very efficient way. After this introduction, the paper presents the general features of the ELAN language. Then we describe the interpretor and the compiler as well as their performances. We also describe the standard library as provided with the system distribution and we give a short description of some of the applications developed in the language such as constraint solving, logic programming or theorem proving.

2 A short description of ELAN

2.1 ELAN *components*

Since we wanted ELAN to be modular, with a syntactic level well-adapted to the user needs, the language is designed in such a way that programming can be done at three different levels:

- First the design of a computational system is done by the so-called *super-user*.
- Such a logic is used by the (standard) *user* in order to write a specification.
- Finally, the *end-user* evaluates queries in the specification, following the semantics described by the logic.

A simple example, which description principle is summarised in Figure 1, is the formalisation of syntactic unification where the tasks are divided as follows:

(i) the super-user describes in a generic way the unification inferences, i.e. a logic for unification, together with a strategy for the application of the inference rules,

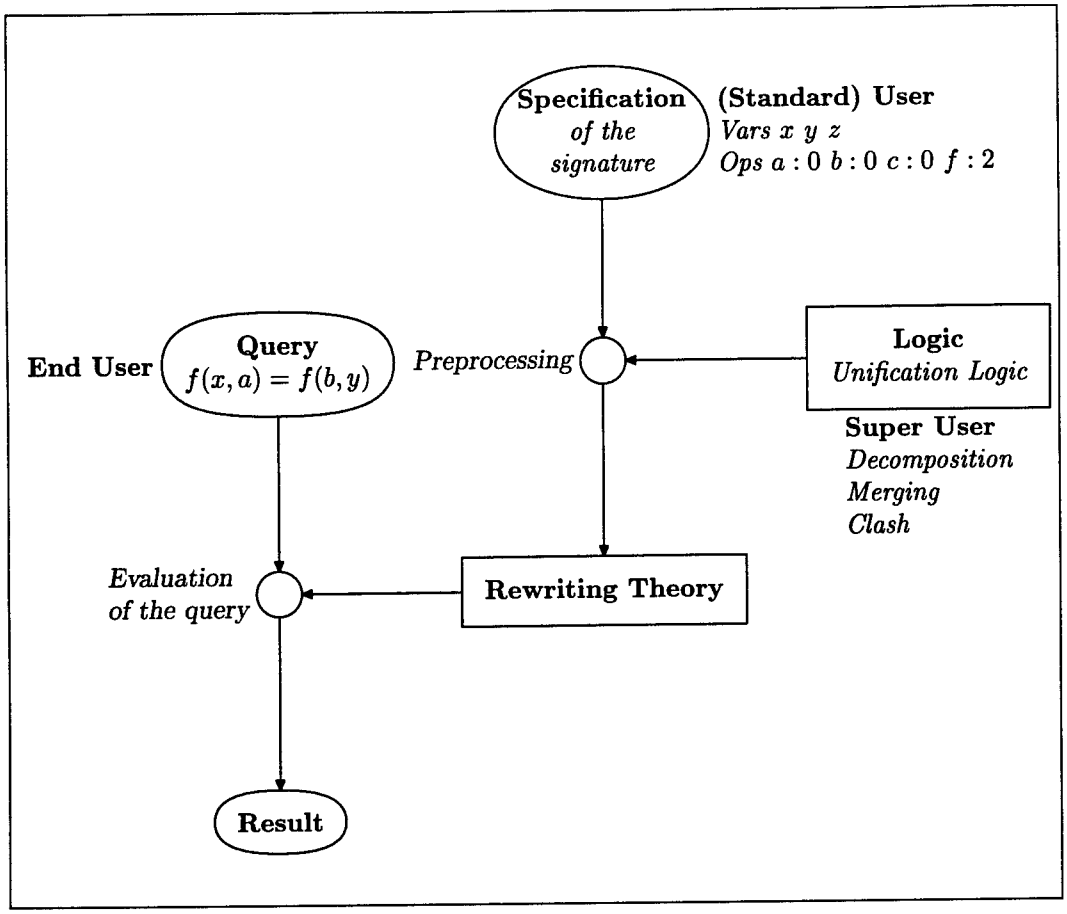


Fig. 1. The representation of syntactic unification in ELAN

(ii) the user gives the specification of an algebra in which (s)he wants to unify terms; in this case, this is quite simple since it amounts to specify the function symbols of the considered term algebra,

(iii) the end-user gives a unification problem.

The description of both the logic and the specification is done in the ELAN syntax, fully described in [KKV95b] and which can be extended by the super-user. In this place, ELAN provides a parser for any context-free grammar. This gives the ability to express grammars in a natural way, and in particular to describe mixfix syntax.

To allow a high degree of modularity, computational system descriptions for the ELAN interpreter are built from elementary pieces that are modules. A module can import other modules and defines its own signature, that is the symbols used to express the syntax of statements and queries. It defines also its own transition/rewrite rules useful to evaluate functions, and its strategies (potentially non-deterministic) for applying these rules. These descriptions are assumed to be done in files with specific extensions: `.lgi` for the top level logic description, `.eln` for a module used in a logic description, `.spc` for a specification (a program written in the defined logic).

To illustrate this, Figure 2 gives the module that the super user has to write in order to describe the derivative operation on simple polynomials. The user

```
LPL polyGeneric description

        // The syntax of the specification that could be given by the user
        // (in ELAN's terminology) is described next. In this example
        // it consists in only one part called 'Vars'
        // This fixes the syntax allowed in the file 'someVariables.spc'.
  specification description
    part Vars        of sort list[identifier]
                     modules varSyntax
  end

  query      of sort polynomial
               result    of sort polynomial
               modules   polyGeneric
               start with (derive)query

end of LPL description
```

Fig. 2. A simple logic example

then gives an actual signature in which he wants to execute the logic. In this simple case described in Figure 2.1, it consists simply to define the variables on which the monomials are built. Notice that the syntax in which the user must describe the specification is fixed by the super user in the description of the current logic. The logic is using modules that contain the description of the

```
specification someVariables

  Vars                X    Y    Z

end of specification
```

Fig. 3. A simple specification example

computational system, i.e. the rewrite rules and the strategies as described for example in Figure 4.

2.2 Strategies

Strategies is one of the main originality of ELAN. Practically, a strategy is a way to describe which computations the user is interested in. A strategy specifies where a given rule should be applied in the term to be reduced. From a theoretical point of view, a strategy is a subset of all proof terms defined by the current rewrite theory. The application of a strategy to a term results in the (possibly empty) set of all terms that can be derived from the starting term using this strategy [KKV95a]. When a strategy returns an empty set of terms, we say that it *fails*.

The current version of the language allows a restricted but still powerful definition of strategies that are built in two steps. The *first level* consists of defining regular expressions built on the alphabet of rule labels. Moreover a rule can be applied using a user-defined strategy only at the top of a term. But this can be combined with a *second level* that consists of using strategies

```
module polyGeneric

            // In the next importation, the role of 'anyIdentifier' is
            // fundamental in order to be able to parse the specification
            // described in the file 'someVariables.spc'
            // and the query, without having to type quotes ("..").
import global
    Vars anyIdentifier list[identifier];

            // We now have to introduce the sort 'variable' in order to
            // make the distinction between constant and variable monomials.
sort variable monomial polynomial ;

op global

            // The next construction 'FOR EACH' is allowed by the ELAN
            // pre-processor, in order to generate automatically a rewrite theory
            // from a given logic and specification.
            // In this exemple, we introduce for each symbol given in the part
            // 'Vars' of the specification a new variable, having the same name.
     FOR EACH Id:identifier SUCH THAT Id:=(listExtract) elem(Vars) :
{ Id : variable; }
  one  : monomial;
  zero : monomial;

            // This just says that a variable is a monomial
  @      :          ( variable ) monomial                                  ;

            // This just says that a monomial is a polynomial
  @      :          ( monomial ) polynomial                                ;
  @ + @  :          ( polynomial polynomial ) polynomial;

endop

rules for polynomial
declare p1, p2, p1p, p2p : polynomial; x : variable;
bodies
  [deriveVar]   x          => one    end
  [deriveOne]   one        => zero   end
  [deriveZero]  zero       => zero   end
  [deriveSum]   p1 + p2    => p1p + p2p   where p1p := (derive)p1
                                          where p2p := (derive)p2
                                          end

end of rules

strategy derive
        dont care choose (deriveVar deriveOne deriveZero deriveSum)
end of strategy

end of module
```

Fig. 4. A simple module example

in the **where** construction of rule definitions. We will see through examples that the expressive power of strategies in ELAN is far more than just regular expressions and that, because of the second level, rules can indeed be applied everywhere in a term. Note also that the next version of ELAN will provide the general strategy definition mechanism described in [BKK96].

The application of a rewrite rule in ELAN yields, in general, several results: i.e. there are several ways to apply a given conditional rule with local assignments. This is first due to equational matching (currently only AC-matching) and second to the **where** assignment, since it may itself recursively return

several possible assignments for variables, due to the use of strategies.

Thus the language provides a way to handle this non-determinism. This is done using the basic strategy operators: **dont care choose** and **dont know choose**.

For a rewrite rule $\ell : l \to r$ the strategy **dont care choose**(ℓ) returns *at most one* result which is undeterministically taken among the possible results of the application of the rule. In practice, the current implementation returns the first one.

On the contrary, if the ℓ rule is applied using the **dont know choose**(ℓ) strategy, then *all* possible results are computed and returned by the strategy. The implementation handles these several results by an appropriate back-chaining operation.

This is extended to the application of several rules: the **dont know choose** strategy results in the application of all substrategies and yields the union of all results; the application of the **dont care choose** strategy returns the set of results of the first non-failing. If all sub-strategies fail, then it fails too, i.e. it yields the empty set.

Two strategies can be concatenated: this means that the second strategy will be applied on all results of the first one. In order to allow the automatic concatenation of the same strategy, ELAN offers the two iterators **iterate** and **while**. The strategy **iterate** corresponds to applying zero, then one, then two, ... n times the strategy to the starting term, until the strategy fails. Thus (**iterate**(s))t returns $\bigcup_{n=0}^{\infty}(s^n)t$. Notice that **iterate** returns the results one by one even when an infinite derivation exists. The strategy **while** iterates the strategy until it fails and return just the terms resulting of the last unfailing call of the strategy. It can thus be defined as (**while**(s))t = $(s^n)t$ where $(s^{n+1})t$ fails.

In order to illustrate how strategies work, let us consider the example consisting of the extraction of the constituents of a list:

```
@: ( elem ) nelist
@ . @: ( elem nelist ) nelist
element(@): ( nelist ) elem

rules for elem
declare   e : elem;
          l : nelist;
bodies
  [extract]  element(e)          => e          end
  [extract]  element(e.l)        => e          end
  [extract]  element(e.l)        => element(l) end
end of rules
```

It we assume furthermore that the constants a,b,c are of sort elem, then:

- (**while dont know choose**(extract) **endwhile**)(a.b.c)
  yields the set {a, b, c},
- (**iterate dont know choose**(extract) **enditerate**)(a.b.c)
  yields the set {element(a.b.c), a, element(b.c), b, element(c), c},
- (**while dont care choose**(extract) **endwhile**)(a.b.c)
  yields the set {a}.

2.3 The execution mechanism

The query is given by the end-user at the ELAN prompt level. There exist two kinds of rules: labelled ones like **deriveSum** in Figure 4 and unlabelled ones, like for instance:

```
[] fib (n)   => fib(n-1) + fib(n-2)   if  n > 1
```

To evaluate a query, the ELAN interpreter repeatedly first normalises the term using unlabelled rules and then applies the transition rules according to the strategy. It works as follows:

Step 1 The current term is normalised using the unlabelled rules. This is done in order to perform functional evaluation and thus it is recommended to the user to provide a confluent and terminating unlabelled rewrite system to ensure termination and unicity of the result. This normalisation process is built-in in the evaluation mechanism and consists in a leftmost innermost normalisation. This should yield always a single result.

Step 2 Then one tries to apply on the normalised term a labelled rule following the strategy described in the logic description. This leads to a (possibly empty) set of terms. If this set is empty, then the evaluation backtracks to the last choice point; if it is not empty, then the evaluation goes on by setting a new choice point and evaluating one of the returned terms by going to **step 1**.

In a slightly more formal way, a rule

$$\ell : l \to d \quad s_1, \ldots, s_n$$

where the s_i are either **where** or **if** expressions, is applied on a term t by:

(i) Matching l against t. This computes a set of substitutions. If this set contains more than two elements, one is chosen and the other ones are stored for possible future backtracking. Let σ be the chosen substitution.

(ii) The evaluation goes on by evaluating the expressions $s_n, \ldots, s_1$, one by one and in this *order* (i.e. from n to 1).

(iii) If s_i is of the form **where** $x_i := (strat_i)t_i$, then one of the results (call it t'_i) of the application of the strategy $strat_i$ on the term t_i is chosen, and the substitution σ is extended by $x_i \mapsto t'_i$. The other results are stored for possible backtracking, and the evaluation goes on with s_{i-1}. If the strategy $strat_i$ fails on t_i, then we backtrack to the previous choice point.

(iv) If s_i is of the form **if** c_i, then the term c_i is evaluated following the normalisation strategy. If the result is the `bool` constant `true`, then one evaluates the next expression s_{i-1}, otherwise one backtracks to s_{i+1}.

This is fully described in [KKV95b] and [Vit94].

3 The ELAN environment

The ELAN language is provided in a system that encompasses an interpretor and a compiler. We give in this section an overview of their features.

3.1 The ELAN *interpretor*

The first way to use ELAN is to run the interpretor using a command like:

```
elan polyGeneric.lgi someVariables.spc
```

Then the user is prompted for a query to be reduced and the results are displayed. A top level command language allows the user to load modules, to run script files, to reduce terms, to display information on the internal state of the system, to trace at the appropriate level the execution of a strategy on a term. The interpretor offers the evaluation of associative and commutative symbols.

3.2 The ELAN *compiler*

An efficient compiler [Vit96] has been designed and implemented. One of its main originality is to allow the efficient execution of non-deterministic rewriting for any ELAN program encompassing no associative-commutative operators. Its efficiency comes mainly from a clever implementation of non-deterministic rewriting, from its memory management and from techniques like many to one matching.

To give an idea on its effectiveness, the following execution results have been obtained on a Sun Ultra Sparc 1 machine with 64MB memory under SUN OS 5.5.

Example	# rules applied	# rewrite per second	user time
Group completion	106.966	534.830	0.2
P5 completion	3.005.747	626.197	4.8
prolog queens8	26.633.873	313.339	75.5
queens8	128.949	1.289.490	0.1
queens10	3.426.635	1.223.798	2.8
primes 40000	8.983.549	6.910.422	1.3
fib33	11.405.773	6.003.038	1.9

The group completion is the rule based description of the Knuth and Bendix algorithm. P5 completion is the completion of a term rewrite system which is a variation of the standard group presentation. The example "prolog queens8" consists in the execution of the *Prolog* program queens8 under the meta description of the Prolog interpretor in ELAN. primes and (naive) fib are implementing the enumeration of bounded prime numbers and the computation of the n*th* Fibonacci number. For comparison, CamlSuperLight, the

latest version of the CAML compiler runs fib 33 in 12.5 seconds.

One can notice that the number of rewrite per second varies in a wide range. This is mainly due to the facts that first, the application of non-deterministic rewrite rules is less efficient due to the associated control and that second, we have only counted the successful rewrite, and for some examples there is a huge number of unsuccessful ones due to the specific form of the rewrite system.

This shows that the rewrite concept can now be implemented in such a way that it becomes quite competitive with either functional programming (when deterministic) or logic programming (when nondeterministic), even when combining together these two concepts.

4 The standard library

In ELAN the user has the possibility to start from nothing and to create his own world, using a non-conditional rewriting logic [1]. Nevertheless in most cases, users are interested in using standard data structures to build their own ones. So we provide several standard useful built-ins described below. We also provide standard objects like terms in an ELAN written library called the "standard library". ELAN can be used without any reference to this library, *except* for what concerns the use of the built-in objects. This library has been designed to be small and as efficient as possible. In particular *no* AC operators is used. The resulting code is more efficient, at the price of sometimes heavier descriptions. But this allows using the current version of the ELAN compiler, with the advantages previously mentioned.

4.1 Built-ins

Booleans

ELAN provides the true and false values and introduces the `bool` module. These two values are built-in and are deeply connected to the implementation of conditions in rewrite rules. To enrich the booleans, *polymorphic* equality and disequality are defined and are also built-in.

Numbers

Numbers can of course be created "by hand", but we choose in ELAN to provide built-in integers and floating point computations. Floating point computations, as provided by the C compiler used for creating your ELAN version, are available using the `double` module.

Identifiers

Two important built-in modules concern identifiers. First the standard ones (i.e. without quotes) and a similar version but with quotes. In fact

[1] Indeed rewriting with conditional rules is connected to the built-in booleans since firing a rule results from a positive match and the evaluation of the condition to the built-in value true.

quoted identifiers are often used by the super user when defining a logic in order to avoid syntactic conflicts at parsing time. Unquoted identifiers are mostly used in specifications.

Elementary term computations

Since they are of primarily use in symbolic computations on terms (and remember that everything in ELAN is a term except the built-ins), several operations like taking the subterm at a given position, or replacing a subterm by another term, are provided as built-ins.

4.2 Standard ELAN *modules*

Based on the above built-ins, the following modules are provided. They are all written in ELAN and are easily modifiable.

Parameterised pairs and parameterised lists are provided with their standard strategies. Terms (with or without variables) are built as a parameterised module that uses its own reference. Note that one difficulty is that the signature is coming from the specification given by the programmer. Substitutions on terms are also provided, as well as equations, system of equations and syntactic unification.

Basic computations on atoms are available in the same spirit as for terms. Finally several modules are given for describing a possible syntax for the user specifications. More complicated syntax (e.g. mixfix) can also be defined.

5 Contributed works

This section surveys several examples that have been fully developed using ELAN. It shows that the rule-based approach of general deduction as presented in [MOM93], as well as more specific processes, like unification advocated for example in [JK91], can be realistically used in order to directly implement these concepts.

5.1 Mini Prolog and narrowing

A simple programming language based on Horn clause logic and SLD-resolution has been implemented in ELAN and is fully described in [Vit94,KKV95a].

In Horn clause logic, formulas are of the form $A \Leftarrow B_1, \ldots, B_n$ with A, B_i being atoms, and a theory is given by a signature and a set of formulas. SLD-resolution is mainly described with two rules that are direct translations of the resolution and reflection rules.

In the same vein, the constraint narrowing process has been described. Thanks to its modularity, it can be easily combined with commutative unification giving in a very simple and elegant way the first (to our knowledge) implementation of commutative narrowing.

5.2 Constraint solving

We have experimented the use of ELAN on many constraint solving mechanisms including syntactic unification (which is provided in the standard library), commutative unification but also disunification as well as certain ordering constraints based on the recursive path ordering.

More recently, ELAN has been used in formalising the consistency techniques used for the constraint satisfaction problem. As described in [Cas96], this provides a nice application of computational systems to the rule-based formalization of these techniques.

Finally let us mention the prototypal use of ELAN in the preprocessing of the finite domain constraints given to the ILOG solver [PL95]. The idea consists in transforming the input constraints into a well-suited representation to allow the best possible propagation by the solver. At this occasion, Gauss elimination, simplification and elementary computations on polynomials have been encoded in ELAN.

5.3 Constraint Solving Combination

Based on theoretical works on the modular combination of constraint solvers (see e.g. [BS95,Rin96a,KR94]), we are now using the capability of ELAN to interact with external programs in order to combine constraint solvers. The main idea developed in [Rin96b] is to incorporate built-in computations that need special data structures to be efficient. Typically, ELAN provides syntactic unification from which commutative unification can be derived just be adding another decomposition rule for commutative symbols. For AC-unification (unification modulo an associative and commutative symbol) the algorithm is quite more complicated, and needs in particular to solve linear Diophantine equations. On the other hand, quite efficient implementations of AC-unification already exist. It is thus natural to use ELAN in order to described the constraint combination logic and then to run the individual constraint solvers, either as built-ins (e.g. for AC-unification) or with their ELAN description (e.g. for commutative unification).

5.4 Completion

One of our initial goal in designing ELAN was to provide a logical framework in order to perform proof of program properties. A step toward this goal is thus to describe completion of a rewrite system in order to be able to perform proof of termination and confluence of equational specifications. This has been done in ELAN using the general approach of deduction with constraints [KKR90]. This allows having an executable description of the deduction process which is the same as the rule-based one commonly used in papers. Furthermore, the flexible strategy description allowed by ELAN gives to the implementer the possibility to experiment various completion approaches. Figure 5 gives an idea of the way we encoded it in ELAN. This is described in [KM95].

```
module constraintCompletion[vars,fss,prec,syst]

...

rules for constraintEquation
declare
        s,t,l,r,g,d                    : term;
        omega                          : list[int];
        newConstraint                  : constraint;
        newConstraintEq                : constraintEquation;
        c,c1                           : constraint;
        sigma                          : substitution;
bodies

   [deduce] deduce(g->d[c1], l->r[c]) => newConstraintEq
        where newConstraintEq     := (deleteVariables) g[r] at omega=d[newConstraint]
        where newConstraint       := (unifys) c & c1 & g at omega =? l
        where omega               := (nvocc) nvocc(g)
   end

...
end of rules


strategy deduce
   dont know choose(deduce)
end of strategy

end of module
```

Fig. 5. A completion rule and strategy

5.5 Higher-order unification

Considering a logical framework like ELAN based on rewriting logic which is essentially first-order, one question arises quickly: "how convenient is such a framework to express higher-order features?". A first answer is given in [DHK95], where unification in the simply typed lambda-calculus is expressed in the first-order equational calculus of explicit substitutions. This has been implemented in ELAN in a quite natural way, both in the general case [Bor95] and in the restricted situation of patterns described in [DHKP96].

6 Conclusion

The ELAN system has been designed and implemented with the general intent to understand the concrete power and usefulness of rewriting for both deduction and computation. This naturally takes place in the current stream of several works on logical frameworks. In our case, the base framework is rewriting logic. The various experiments made possible by the existing implementation have shown the fundamental interest of such a logical framework at the edge between logic and computation. The powerful notion of strategy, that we have designed and implemented, allows us to describe, in the potential search space, the deductions that one wants to explore, and to guide the computations so that they become more efficient.

This work and many others stemming from the seminal idea of rewriting logic lead the way to an exciting research field, where many general questions

are arising, among which we can mention:

- the use of computational systems for the design of an integrated proof environnement where the programs, the proofs, the provers and their proof plans, but also specific decision procedures, can all be designed and executed in the same uniform framework,
- the use of rewriting logic for inductive reasoning,
- the reflective power of rewriting logic,
- the complexity of mapping from a given logic to rewriting logic,
- the relationships between linear and rewriting logics,
- the comparison with other logical frameworks[2].

The work on term rewriting, started now 25 years ago, allowed very strong theoretical interests and results to emerge. But it has encountered sceptiscism because of the relative inefficiency of rewriting as a computation and a fortiori as a deduction process. One main originality of ELAN is its capability to perform deterministic and non-deterministic computations, thanks to the dont-know and dont-care choose operators on strategies. The current implementation of ELAN with in particular its compiler, shows that the inefficiency criticism is no more valid, since deterministic as well as undeterministic rewriting can be executed as fast as the best compiled logical languages like ML or CLP. We get now the evidence that rewriting is combining the descriptive and computational power needed for many applications in a unique yet very simple concept. Thus, because of the crucial role played by rewriting in this context, many of the questions studied during the last decades surface again under the more genral point of view of transition systems, since the intended semantics is no more restricted to the (up to now) standard equational interpretation.

From the more specific point of view of ELAN, several works are under development. Let us mention some of them.

- A system for performing input/ouput is currently being designed [Vir96].
- We are investigating the extension of the language, and its interpretor and compiler, to the order-sorted rewriting case with built-in theories like associativity and commutativity.
- Since quite complicated proofs can be achieved using the system (for example the full trace of the execution of the completion process mentioned above could be as big as several 100 Mo), appropriate tools for editing and understanding such proofs are under design.
- Internalizing the proof terms and the strategies in the logic is an extremely powerful idea that allows in particular to define strategies using the concept of computational system. Investigations in this direction are described in [BKK96].
- A first indication of the reflective power of a framework is to be able to express its own evaluation. In ELAN, it is possible to express in a very

[2] See the web-page at `http://www.cs.cmu.edu/afs/cs/user/fp/www/lfs.html`

natural way the rewriting process itself [Vit94,KKV95a]. We are now investigating the reflective power of the rewriting logic in this context and the introduction of reflective capabilities in ELAN, as proposed in [KM96].

- Rewriting logic is a natural framework to express concurrency, but conversely it it quite challenging to use concurrent rewriting to efficiently execute deductions and computations. We have first investigated this line of research in [KV90], designing an implementation of fine grained concurrent rewriting together with its garbage collector [Alo95] for unconditional as well as conditional rewriting [AK96]. The implementation ReCo runs on various parallel architectures and we currently investigate the definition of specific strategies for concurrent rewriting.

Further information on the ELAN system, including the current distribution of the system, can be found at the following address:
`http://www.loria.fr/equipe/protheo.html/PROJECTS/ELAN/elan.html`.

Acknowledgment: We thank for their interaction in the design and the implementation of ELAN, Jounaidi Benhassen, Carlos Castro, Christophe Ringeissen, Patrick Viry and the Protheo team. We also thank José Meseguer for our long collaboration on many topics concerning the theoretical background as well as the implementation of ELAN.

References

[AK96] I. Alouini and C. Kirchner. Toward the concurrent implementation of computational systems. In M. Hanus, editor, *Proceedings of ALP'96*, Lecture Notes in Computer Science, Aachen (Germany), September 1996. Springer-Verlag.

[Alo95] I. Alouini. Concurrent garbage collection for concurrent rewriting. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 132–146. Springer-Verlag, 1995.

[BKK96] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[Bor95] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In M. Bartošek, J. Staudek, and J. Wiedermann, editors, *Proceedings of the SOFSEM'95: Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 363–368. Springer-Verlag, 1995.

[BS95] F. Baader and K. U. Schulz. Combination of constraint solving techniques: An algebraic point of view. In J. Hsiang, editor, *Rewriting Techniques and Applications, 6th International Conference, RTA-95*, LNCS 914, pages 352–366, Kaiserslautern, Germany, April 5–7, 1995. Springer-Verlag.

[Cas96] C. Castro. Solving Binary CSP using Computational Systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[DHK95] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions, extended abstract. In D. Kozen, editor, *Proceedings of LICS'95*, pages 366–374, San Diego, June 1995.

[DHKP96] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of JICSLP'96*, Bonn (Germany), September 1996. The MIT press.

[GKK+87] J. A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[KKR90] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on Automatic Deduction.

[KKV95a] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, pages 131–158. The MIT press, 1995.

[KKV95b] C. Kirchner, H. Kirchner, and M. Vittek. ELAN *V 1.17 User Manual.* Inria Lorraine & Crin, Nancy (France), first edition, November 1995.

[KM95] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.

[KM96] H. Kirchner and P.-E. Moreau. A reflective extension of ELAN. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[KR94] H. Kirchner and C. Ringeissen. Combining symbolic constraint solvers on algebraic domains. *Journal of Symbolic Computation*, 18(2):113–155, 1994.

[KV90] C. Kirchner and P. Viry. Implementing parallel rewriting. In P. Deransart and J. Maluszynski, editors, *Proceedings of PLILP'90*, volume 456 of *Lecture Notes in Computer Science*, pages 1–15, Linköping (Sweden), August 1990. Springer-Verlag.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MOM93] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[PL95] J.-F. Puget and M. Leconte. Beyond the black box,: Constraints as objects. In J. Loyd, editor, *Proceedings of the International Logic Programming Symposium*, pages 513–527. The MIT press, 1995.

[Rin96a] C. Ringeissen. Combination of matching algorithms. *Information and Computation*, 1996.

[Rin96b] C. Ringeissen. Prototyping combination of unification algorithms with ELAN. Technical report, CRIN, 1996.

[Vir96] P. Viry. Input/output for rule based languages. Technical report, Universita di Pisa, 1996.

[Vit94] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, October 1994.

[Vit96] M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), July 1996. Springer-Verlag.

Electronic Notes in Theoretical Computer Science 4 (1996)

Input/Output for ELAN

Patrick Viry

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56100 Pisa, Italy
Email: viry@di.unipi.it

Abstract

We show how to add Input/Output capabilities to the ELAN rewriting interpreter using a rewrite specification of π-calculus. This I/O system has the advantage of being totally explicit and fit in the same semantic framework than any other "application program". An actual implementation shows the effectiveness of this approach.

1 Introduction

Currently available rewrite interpreters (e.g. OBJ3 [GKK+87], Redux [Bün93], ELAN [KKM95]) offer a nice programming model and are quite efficient, but lack input/output (I/O) capabilities.

For other models of computation like functional or logic programming, the typical approach has been to add extra features implementing I/O. But these features do not fit in the nice and simple underlying model: in order to be able to understand or reason about programs involving I/O, the basic computational model has to be extended in non trivial ways, making it not so nice and simple anymore (see for instance [Gor94] about the monadic approach to functional I/O). The problem is that functional computation and I/O have two different and a priori incompatible interpretations, one in terms of equality and the other in terms of transitions between states.

In the case of rewriting, the underlying semantic model, called rewriting logic [Mes92,MOM93], allows to combine in a simple way equational computations such as function evaluation or abstract data types and non equational computations such as transitions between states.

This is fine on a theoretical point of view, but a straightforward implementation is not realistic, since it would imply matching modulo an arbitrary big equational theory. We have shown in [Vir95] that an effective implementation is possible in many common cases, by orienting equations of the equational theory into rewrite rules, thus ending up with two sets of rules:

[1] Supported by an HCM fellowship, EuroFOCS Network

© *1996 Elsevier Science B. V.*

the reduction rules (denoted with $\longrightarrow$) describe transitions between states and the equational rules (denoted with $\xrightarrow{=}$) implement the equational theory. We introduced three notions of coherence and proved that they are sufficient conditions for three notions of equivalence to hold between the intended semantics of rewriting logic and the actual implementation. Strong coherence is the strongest property, it implies an exact correspondence between steps by reduction rules, but is difficult to assess in the presence of non linear equational rules (rules where more than one occurrence of a variable appears in the left-hand or right-hand side). Equational coherence is the weakest property, and only implies preservation of normal forms. In the middle, weak coherence implies preservation of derivations.

The coherence properties can be verified by checking critical pairs between rewrite rules, and in the weak and strong cases ensuring that a non linear equational rule can never be applied above a reduction rule. The reader is referred to [Vir95] for more details.

In this paper, We take advantage of this result to design and implement an I/O model for rewrite interpreters that will be *totally explicit* in the *same* framework as any other programs.

The model we propose is based on π-calculus [Mil91], a well studied calculus exhibiting processes exchanging messages.

Process calculi are traditionally described by their transition relation $P \xrightarrow{\alpha} Q$ (the process P is able to "perform the action" α and then behave as Q). Implementing this relation by rewriting is not trivial, since arrows of the transition relation are labelled and rewrite steps are not, and attempts to do so require unnatural tricks in order to take labels into account [MOM93].

However, the transition relation is not what we need. It is useful for understanding process equivalence, based on external observation, but we are interested here in the internal steps of process behaviour, described by the so-called reduction relation. In fact, *it is enough to implement internal transitions steps in order to realize an I/O system*,: an external communication is nothing else than an internal communication in a wider context, and the two relations can be defined in terms of each other (see section 2.4).

In a previous paper [Vir96], we gave a rewriting definition of the reduction relation of π-calculus and formally proved its correctness. We first start with recalling this definition, referring to [Vir96] for formal proofs. We then show how it can be used as a basis for adding input/output capabilities to ELAN [Vit94,KKM95], present our implementation and explain design choices. We finally give some examples of programs using this approach.

The source code of the implementation is available from the author.

2 π-calculus and its reduction relation

2.1 Why π-calculus

When choosing a calculus in which to specify explicitly input-output, we first have to decide on a conceptual model. The typical choice is to consider *processes*, able to evolve independently and to synchronize or exchange data by message passing through designated *channels*. This models fits quite well the intuition of sets of boxes connected between themselves and to the outside by some wires.

But then one may argue that π-calculus is *conceptually* too complex for such a "simple" task, and that either an ad-hoc calculus or a *conceptually* simpler calculus such as CCS with value passing or LOTOS [BB89] may do the job.

The first idea is that a special-purpose calculus expressive enough will anyway exhibit all the complexity of π-calculus, hence it is better to rely on a known calculus whose semantic foundation has been well studied. Now why π-calculus? Is mobility of processes (the ability to dynamically change a configuration) really needed?

The second idea is that, although introducing mobility makes a calculus more complex on a semantic point of view, *mobility comes for free* on an operational point of view. In order to *explicitly* model exchange of data, a notion of *name* (corresponding e.g. to the bound variables of λ-calculus) together with a notion of explicit substitution are needed. But then there is no difference between, say, substituting an integer value for a name, and substituting a channel name for another name. Or the other way round, we may say that value-passing comes for free when having mobility.

By acknowledging the fundamental role of names and explicit substitutions, we end up with a simpler calculus, in which mobility and exchange of data are modeled by the *same* set of rules.

2.2 π-calculus with explicit substitutions

In this section and the following, we briefly introduce a rewriting implementation of the reduction relation of π-calculus. More details and proofs of the correspondence results can be found in [Vir96].

In the original definition of π-calculus, substitution is a meta-operation, not part of the calculus. But for an actual implementation the substitution operation has to be made explicit, by introducing a substitution operator and the rules defining it (often referred to as the "substitution calculus").

The terms of π-calculus are usually terms with higher-order variables, considered up to α-conversion, that may be bound by binder operators (note that these higher-order variables are different from term variables, and are represented by constant names).

It is possible to design a substitution calculus for terms with higher-order variables (see e.g. [MOM93]), but the resulting calculus is very inefficient,

mainly because one often needs to check if a variable is free or not. Efficient substitution calculi are based on terms with so-called De Bruijn indices, where an higher-order variable is replaced by an integer indicating how many binders to jump over until finding the binder associated with that variable. The change of representation is transparent as there is a one-to-one relationship between well-formed terms with indices and terms with higher-order variables not containing free names. For instance, in the case of λ-calculus, the term $\lambda x.\lambda y.(xy)$ would be represented as $\lambda_x.\lambda_y.(\underline{\mathbf{1}}_x\underline{\mathbf{0}}_y)$ (the subscripts x and y are not actually part of the term, we add them here and in the following for better readability).

Substitution calculi based on terms with indices are more efficient and very close to actual machine implementations. In fact, the different machines designed for implementing functional reduction can be seen as different strategies of applying the substitution rules [HMP95]. In our implementation, we use a substitution calculus inspired by $\lambda\upsilon$-calculus [LRD94], which is one of the simplest given in the literature. Refer to [Les94] for a survey on the various substitution calculi.

The terms of π-calculus with indices are defined as follows (using a syntax more digestible to rewrite interpreters than the usual one) :

- indices are integers, written always underlined, like $\underline{\mathbf{3}}$ or $\underline{\mathbf{n+1}}$. For better readability, we usually add a subscript with a variable name to both indices and binders, as in $(\nu)_x\mathsf{in}(\underline{\mathbf{0}}_x).\mathsf{nil}$.

- processes

$P :=$	nil	the inactive process
	$\mid (\nu)\,P$	restriction (binder)
	$\mid g.P$	guard (see below)
	$\mid P_1 + P_2$	choice
	$\mid P_1 \mid P_2$	parallel composition
	$\mid\, !\,P$	replication
	$\mid P\sigma$	substitution (see below)

- guards

$g :=$	$\mathsf{in}(c)$	input on channel c (binder)
	$\mid \mathsf{out}(c, x)$	output x on channel c
	$\mid \mathsf{bout}(x)$	bound output (binder)
	$\mid \tau$	internal choice

  Bound output can be defined in terms of other basic operators ($\mathsf{bout}(x).P = (\nu)_c\mathsf{out}(\underline{\mathbf{0}}_c, x).P$), but we choose to introduce it explicitly for technical reasons.

- substitutions

$$\begin{aligned}\sigma := \quad & [a/] \\ | \quad & \Uparrow(\sigma) \\ | \quad & [\uparrow]\end{aligned}$$

The intuitive meaning of the substitution operators is to map indices as follows:

$[a/]$	$\Uparrow(s)$	$[\uparrow]$
$\underline{\mathbf{0}} \mapsto a$	$\underline{\mathbf{0}} \mapsto \underline{\mathbf{0}}$	$\underline{\mathbf{0}} \mapsto \underline{\mathbf{1}}$
$\underline{\mathbf{1}} \mapsto \underline{\mathbf{0}}$	$\underline{\mathbf{1}} \mapsto s(\underline{\mathbf{1}})[\uparrow]$	$\underline{\mathbf{1}} \mapsto \underline{\mathbf{2}}$
...	...	...
$\underline{\mathbf{n+1}} \mapsto \underline{\mathbf{n}}$	$\underline{\mathbf{n+1}} \mapsto s(\underline{\mathbf{n}})[\uparrow]$	$\underline{\mathbf{n}} \mapsto \underline{\mathbf{n+1}}$

Processes are considered modulo the equations AC(+) and AC(|) (associativity and commutativity of the + and | operators) and the following equational rules:

$$\begin{aligned} P + \mathsf{nil} &\xrightarrow{=} P \\ P \mid \mathsf{nil} &\xrightarrow{=} P \\ !P &\xrightarrow{=} P \mid !P \\ P \mid (\nu)\, Q &\xrightarrow{=} (\nu)\,(P[\uparrow] \mid Q) \end{aligned}$$

Another set of equational rules deals with the application of explicit substitutions. They are inspired by the rules of $\lambda\upsilon$-calculus [LRD94], extended to take into account the three different binding operators present in π-calculus (refer to [Vir96] for details), There are basically two kind of rules, the congruence rules "pushing down" the substitution operators, and the variable substitution rules.

The last two equational rules allow to replace | or ! operators at the top of a process with disjunctions:

The expansion rule. Let $P = \alpha_1.P_1 + ... + \alpha_n.P_n$ and $Q = \beta_1.Q_1 + ... + \beta_m.Q_m$, then

$$P \mid Q \quad \xrightarrow{=} \quad \sum_{i=1...n} \alpha_i.(P_i \mid Q) + \sum_{i=1...m} \beta_i.(P \mid Q_i) + \sum_{\substack{i=1...n \\ j=1...m}} Sync(\alpha_i.P_i, \beta_j.Q_j)$$

where

$$Sync(\alpha_i.P_i, \beta_j.Q_j) = \begin{cases} \tau.(P_i[z/] \mid Q_j) & \text{if } \alpha_i = \mathsf{in}(x) \text{ and } \beta_j = \mathsf{out}(x, z) \\ \tau.(\nu)\,(P_i \mid Q_j) & \text{if } \alpha_i = \mathsf{in}(x) \text{ and } \beta_j = \mathsf{bout}(x) \\ \text{(and similarly by swapping the arguments)} & \\ \mathsf{nil} \text{ in all other cases} & \end{cases}$$

The replication rule. Let $P = g_1.P_1 + \cdots + g_n.P_n$, then

$$!P \quad \xrightarrow{=} \quad g_1.(P_1 \mid !P) + \cdots + g_n.(P_n \mid !P)$$

A process is in weak disjunctive normal form if is of the form

$(\nu)\ldots(\nu)(g_1.P_1 + \cdots + g_n.P_n)$

with $n \geq 0$. Using the above equational rules, any process can be put in weak disjunctive normal form.

Expansion and replication are not properly rewriting rules because of the variable n, but can be easily simulated using extra hidden operators.

The rewrite relation defined by the above equational rules (modulo AC) does not terminate because of the replication rule that can be applied repeatedly into its own right-hand side. In order to ensure termination, this latter rule has to be applied only when needed in order to compute a weak disjunctive normal form (see [Vir96] for a precise definition). Let us denote $\Longrightarrow$ the derivations using all the above equational rules (modulo AC) according to that strategy, then we have the following correspondence result:

Proposition 2.1 ([Vir96]) *There is a one-to-one correspondence between processes of the original π-calculus (modulo the usual structural axioms) and normal forms with respect to $\Longrightarrow$.*

In the following, $=$ will denote equivalence of processes modulo $\Longrightarrow$.

2.3 The reduction relation

The reduction relation of π-calculus, written $\longrightarrow$, corresponds to internal transitions of processes (the so-called τ-transitions). It is defined as

$$\tau.P + Q \longrightarrow P \qquad \text{(Choice)}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \frac{P \longrightarrow P'}{(\nu)\,P \longrightarrow (\nu)\,P'}$$

The Choice rule is not a rewrite rule, since it cannot by applied under any context, but it can be considered a rewrite rule if used together with a strategy that permits its application only under the allowable contexts. Choice is the only non equational rewrite rule (denoted with $\longrightarrow$ rather than $\stackrel{=}{\longrightarrow}$): it is interpreted as an irreversible transition between states, whereas all the above equational rules only compute an equivalent form of a given process.

In the following we denote $\longrightarrow$ the rewrite steps applying Choice considered as a rewrite rule with the appropriate strategy. The correspondence result is as follows:

Proposition 2.2 ([Vir96]) *There is a reduction step $P \longrightarrow Q$ in the original π-calculus if and only if there is a rewrite derivation $P \Longrightarrow\!\!\longrightarrow Q'$, where Q' is structurally equivalent to Q.*

This result is proved by showing strong coherence between equational and reduction rules and applying the results of [Vir95].

The relations $\Longrightarrow$ and $\longrightarrow$ are based on rewriting modulo AC and can be implemented quite efficiently in ELAN.

2.4 *Internal vs. External Communications*

The reduction relation of π-calculus describes internal moves of processes but is not able to take into account external communications. However, an external communication between a process P and an environment E (another π-calculus process) is nothing else than an internal communication within the combined process $P \mid E$ (at the top level, ruling out communications within P or within E).

Observation of the external communications of P can be defined as predicates as follows:

Definition 2.3 *The observation predicates are defined as:*

$$P \xrightarrow{\mathsf{in}(c)} P' \textit{ iff. } P = (\nu)...(\nu)(\mathsf{in}(c).P' \mid Q)$$

$$P \xrightarrow{\mathsf{out}(c,x)} P' \textit{ iff. } P = (\nu)...(\nu)(\mathsf{out}(c,x).P' \mid Q)$$

The relevance of this definition is stated by the following property:

Proposition 2.4 *There is a step of the reduction relation $P \longrightarrow Q$ if and only if P is of the form $P = (\nu)...(\nu)(P_1 \mid P_2)$, Q is of the form $Q = (\nu)...(\nu)(Q_1[x/] \mid Q_2)$, with $P_1 \xrightarrow{\mathsf{in}(c)} Q_1$ and $P_2 \xrightarrow{\mathsf{out}(c,x)} Q_2$.*

A process P can communicate with an environment E if $P \xrightarrow{\mathsf{in}(c)} P'$ and $E \xrightarrow{\mathsf{out}(c,x)} E'$, or the opposite. Property 2.4 basically says that there P can communicate with E if and only if there is an internal reduction $P \mid E \longrightarrow P' \mid E'$: any transition can be considered equivalently as an internal transition or an external communication, depending what we consider as the external environment.

The whole implementation of I/O is based on this idea: even for implementing external communication, it is enough to implement internal transitions.

Proof of proposition 2.4

- Only if part:

  Consider $P = (\nu)...(\nu)(P_1 \mid P_2)$. If $P_1 \xrightarrow{\mathsf{in}(c)} Q_1$, then P_1 is of the form $P_1 = \mathsf{in}(c).Q_1 + U_1$. Similarly $P_2 = \mathsf{out}(c,x).Q_2 + U_2$. Then using the expansion rule, we have $P = (\nu)...(\nu)(\tau.(Q_1[x/] \mid Q_2) + \mathsf{in}(c).(Q_1 \mid \mathsf{out}(c,x).Q_2 + U_2) + \mathsf{out}(c,x).(\mathsf{in}(c).Q_1 + U_1 \mid Q_2))$, and the Choice rule applies giving $P \longrightarrow (\nu)...(\nu)(Q_1[x/] \mid Q_2)$.

- If part:

  If $P \longrightarrow Q$, then by definition of the reduction relation and the equation $U \mid (\nu)\, V \xrightarrow{=} (\nu)\,(U[\uparrow] \mid V)$, P must be of the form $P = (\nu)...(\nu)(\tau.Q + V)$. The τ symbol can only be introduced by the expansion rule, hence P is of the form $P = (\nu)...(\nu)(\tau.(Q_1[x/] \mid Q_2) + \mathsf{in}(c).(Q_1 \mid \mathsf{out}(c,x).Q_2 + U_2) + \mathsf{out}(c,x).(\mathsf{in}(c).Q_1 + U_1 \mid Q_2)) = \mathsf{in}(c).Q_1 + U_1 \mid \mathsf{out}(c,x).Q_2 + U_2$, thus $P = P_1 \mid P_2$ with $P_1 \xrightarrow{\mathsf{in}(c)} Q_1$ and $P_2 \xrightarrow{\mathsf{out}(c,x)} Q_2$, and $Q = Q_1 \mid Q_2$.

3 Implementation

ELAN [KKM95] is a rewrite interpreter and compiler developed in Nancy, with a strong emphasis on efficiency. Two of its specific features are of particular interest to us:

- It offers a powerful means of defining strategies, which makes it easy to define the Choice rule as a rewrite rule that can be applied only under some contexts, and to encode the lazy application of the replication rule.
- It has a powerful preprocessor that we use for defining rule schemata for the substitution rules, since they must apply to any variable type.

The relations $\Longrightarrow$ and $\longrightarrow$ are thus easily encoded into ELAN.

3.1 Communication scenario

A process in weak disjunctive normal form exhibits the possible external and internal communications:

$$P = \alpha_1.P_1 + \cdots + \alpha_n.P_n + \tau.Q_1 + \cdots + \tau.Q_n$$

The subterms $\alpha_i.P_i$ offer possible external communications, that can be "performed" if the environment accepts them, namely if the corresponding Unix file descriptors are ready for input or output.

The subterms $\tau.Q_j$ correspond to possible internal choices and can possibly be "selected" by applying the Choice rule.

The problem is what to do with a process P containing both $\alpha_i.P_i$ and $\tau.Q_j$ subterms. We can imagine three scenarios:

(i) Perform an internal transition by applying the Choice rule to one of the $\tau.Q_j$

(ii) Perform an external communication, waiting as long as necessary until one is accepted.

(iii) Check if one of the external communications is accepted, if yes perform it, if no perform an internal transition

In the first case, the problem is that applying the Choice rule may not terminate. There exist so-called divergent processes that can perform infinitely many internal moves.

In the second case, the implementation would not be "fair", in the sense that the program may block indefinitely even if a communication would have been possible after performing an internal move.

The third case avoids both these problems, but raises an issue of efficiency since checking if file descriptors are ready for input or output is a costly operation.

We opted for the first scenario in our implementation, leaving to the user the task of ensuring that there are no divergent processes. This choice is motivated by the feeling that nobody would ever want to design divergent processes, and that we may safely consider this case as an error.

3.2 Actual input/output

Reduction to normal form thus computes a term of the form

$$P = \alpha_1.P_1 + \cdots + \alpha_n.P_n$$

where all the α_i's are input or output guards referring to an external channel.

The selection of a particular external communication among all possible ones (selection of one of the α_i) is implemented by adding a new built-in to ELAN, calling the Unix primitive `select`. Given a set of file descriptors (streams) as arguments, `select` returns the ones that are ready for reading and/or writing.

Effective input/output is then performed by implementing the in and out predicates with the corresponding `read` and `write` Unix system calls. This is done by adding another two new built-in operators in the ELAN source code.

Reduction then proceeds again starting from P_i if $\mathsf{out}(c, x).P_i$ had been selected, or from $P_i[x/]$ if $\mathsf{in}(c).P_i$ had been selected and x is the value read. The whole program stops if the normal form nil is reached, indicating no more possible communication.

3.3 Unix interface

Some adaptations of the program have been necessary in order to cope with "real" input/output:

(i) Since external channels correspond to Unix file descriptors, rather than maintaining a table of associations between π-calculus channels and descriptors, we choose to introduce a special constructor for external channels, $\mathsf{extchan}(i)$, where i is the file descriptor.

This also makes possible the addition of a simplification rule: since an input or output guard on an internal channel may never interact with the outside, we may safely remove processes with such guards from the weak disjunctive normal form computed by $\Longrightarrow$.

(ii) A Unix file descriptor must be opened before its use. We introduced in the calculus a new guard for this purpose, $\mathsf{open}(\text{filename}, \text{type}, \mathrm{P_{Succ}}, \mathrm{P_{Fail}})$, with the intuitive meaning of opening the file whose name is given, then behave as $P_{\mathrm{Succ}}[\mathsf{extchan}(\underline{\mathbf{i}})/]$ if the opening succeeded, binding $\underline{\mathbf{0}}$ with the corresponding channel, or behaving as $P_{\mathrm{Fail}}[i/]$ in case of failure, binding $\underline{\mathbf{0}}$ with a system error number. The type argument is used to convert between ELAN and Unix data representation (for instance ELAN integers may be encoded in Unix files as bit fields of various lengths, or even as their printable representation).

The rules for the expansion and replication theorem must be extended in order to take this new guard into account.

We did not address in this stage of prototype the possibility of closing files, but this would be needed as well for practical applications.

(iii) The semantics of communication in π-calculus and of input-output in Unix are quite different. The former is synchronous and atomic, the latter is asynchronous and may fail in the middle of a transfer.

Asynchronicity is actually not a problem, because it is internal to Unix: the semantic of a communication between a process P and the Unix environment is preserved. Atomicity is guaranteed when reading or writing one byte at a time. It is possible to simulate atomicity for bigger transfers as well, but at the expense of efficiency. A real I/O system should certainly provide a choice between these both options.

Possible failure is more problematic because it does not fit in the model of π-calculus. A possible approach would be to extend input and output guards to allow for failure in a way similar to the open guard, at the expense of simplicity in designing processes. Another more "practically" satisfying approach may be to add a notion of exception handling to the calculus, but this seems a non trivial task. For the moment we simply suppose that such events will make the whole program fail (gossipers note: this is not much different from most commercially available software...)

3.4 Value-passing and typing

So far, we have considered the plain monadic π-calculus. As shown in [Mil91], this calculus is powerful enough to encode any kind of structured data, so the game may end here. However, implementing a value passing calculus based on this encoding loses one of the main advantages of rewriting logic, namely the fact of being able to combine various calculi, using for each domain the more adequate calculus without having to do unnatural encodings.

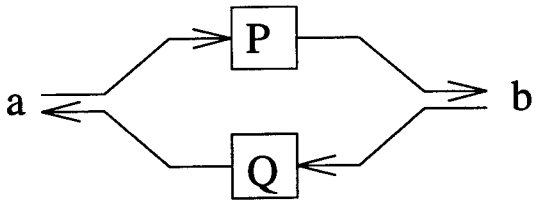
The solution is immediate. Extend the calculus by letting input and output values be not only π-calculus channels, but also any arbitrary data type (indices now range over arbitrary values, including channels). One may for instance write a process $\mathsf{in}(\underline{\mathbf{0}})_x.\mathsf{out}(\underline{\mathbf{1}}_c, 3 + \underline{\mathbf{0}}_x).\mathsf{nil}$, whose behaviour is intuitively "receive an integer value x on a channel c, add it 3 and send the result back on the same channel.

In order to have a conceptual difference between channels and values, some notion of *typing* is called for. This notion of typing should also be able to take into account a possible typing of values.

However, well-typedness cannot be expressed in the many-sorted framework of ELAN, because the type of an index should depend on its value. In our prototype we rely on the user to provide well-typed terms. Classical techniques borrowed from functional programming may be used to guarantee well-typedness.

4 Examples

4.1 Double-way buffer



A double-way buffer consists of two processes in parallel, each of them repeatedly reading a data element from a channel and writing it on the other channel. These processes are most naturally specified using recursive equations:

$$P = \mathsf{in}(a)_x.\mathsf{out}(b, \underline{0}_x).P$$

$$Q = \mathsf{in}(b)_x.\mathsf{out}(a, \underline{0}_x).Q$$

We cannot implement directly recursive equations, and need to restate this definition using the replication operator. Note that this is always possible [Mil91] and that the two specifications are weak equivalent (i.e. equivalent up to the internal actions). P and Q are redefined as

$$P \stackrel{def}{=} (\nu)_p(\mathsf{out}(\underline{0}_p, \mathrm{void}).\mathsf{nil} \mid !\,\mathsf{in}(\underline{0}_\mathrm{p}).\mathrm{in(a)_x.out(b,}\,\underline{0}_\mathrm{x}).\mathrm{out}(\underline{1}_\mathrm{p}, \mathrm{void}).\mathsf{nil})$$

$$Q \stackrel{def}{=} (\nu)_q(\mathsf{out}(\underline{0}_q, \mathrm{void}).\mathsf{nil} \mid !\,\mathsf{in}(\underline{0}_\mathrm{q}).\mathrm{in(b)_x.out(a,}\,\underline{0}_\mathrm{x}).\mathrm{out}(\underline{1}_\mathrm{q}, \mathrm{void}).\mathsf{nil})$$

where void is the only value of the single-valued type of channels that only exchange synchronizations. Intuitively, the process below the replication operator of P (resp. Q) can only be "activated" by an input from channel p (resp. q).

The process $P \mid Q$ reduces to the weak disjunctive normal form

$$(\nu)_p(\nu)_q(\mathsf{in}(a)_x.P' + \mathsf{in}(b)_x.Q')$$

with

$$\begin{aligned} P' = {} & \mathsf{out}(b, \underline{0}_x).\mathsf{out}(p, \mathit{void}).\mathsf{nil} && (1)\\ \mid {} & \mathsf{in}(b)_x).\mathsf{out}(a, \underline{0}_x).\mathsf{out}(q, \mathit{void}).\mathsf{nil} && (2)\\ \mid {} & !\,\mathsf{in}(\underline{0}_p).\mathsf{in}(a)_x.\mathsf{out}(b, \underline{0}_x).\mathsf{out}(\underline{1}_p, \mathit{void}).\mathsf{nil} && (3)\\ \mid {} & !\,\mathsf{in}(\underline{0}_q).\mathsf{in}(a)_x.\mathsf{out}(b, \underline{0}_x).\mathsf{out}(\underline{1}_q, \mathit{void}).\mathsf{nil} && (4) \end{aligned}$$

and similarly for Q' by swapping p with q and a with b.

The normal form exhibits the two possible external communications $\mathsf{in}(a)$ and $\mathsf{in}(b)$. As soon as one of them is possible, the communication takes place and the computation proceeds with either P' or Q'. In P', term (1) is the continuation of the buffer process P, term (2) is the buffer process Q, and terms (3) and (4) are the "pools" of processes that are activated by an input on channel p or q.

The two-way buffer example can be trivially extended by adding compu-

tation of the output values, for instance

$$P = \mathsf{in}(a)_x.\mathsf{out}(b, f(\underline{0}_x)).P$$

$$Q = \mathsf{in}(b)_x.\mathsf{out}(a, g(\underline{0}_x)).P$$

where f and g are functions defined by rewrite rules. Since these defined operators appear only strictly below process operators, we are guaranteed that strong coherence is preserved [Vir95] and thus that our implementation remains correct.

4.2 Filter

The previous example exhibits the typical programming style of our approach : processes exchanging data, possibly computing output values with defined functions. But process expressions may also appear below defined symbols, as long as no non-linear rewrite rule may ever be applied above a process expression, in order to preserve strong coherence [Vir95].

This is the case for instance with the *if...then...else...* operator, defined by the following linear rules

$$\text{if true then } x \text{ else } y \longrightarrow x$$

$$\text{if false then } x \text{ else } y \longrightarrow y$$

Then we can write a FILTER process, that repeatedly inputs values on a channel i and copies them on an input channel o only when they verify a given condition :

$$\text{FILTER} = \mathsf{in}(i)_x.\text{if } c(x) \text{ then } \mathsf{out}(o, \underline{0}_x).\text{FILTER else FILTER}$$

This recursive definition is then restated using the replication operator as in the previous example.

This possibility allows for a more "natural" programming style, for instance closer to CSP/Occam [Hoa78], but the constructs that may appear above processes must be clearly identified in order to ensure the condition about non linear rules.

These constructs may also be the constructors of data types, and we may be able for instance to specify in a unique framework a system of windows each running its own independent process.

5 Conclusion

Starting from a rewriting definition of the reduction relation of π-calculus, we have designed an input/output system for the ELAN rewriting interpreter that is totally explicit in the rewriting framework itself and integrates smoothly with any other "application program".

This system has been implemented in ELAN to show its effectiveness. An important issue yet to be checked is the strategy used for applying the substitution rules. Applying them eagerly is hopelessly inefficient, but particular

strategies correspond to different types of known abstract machines [HMP95] and can achieve the same efficiency once compiled.

We plan to add this system to a future ELAN distribution, and hope that adding I/O capabilities to rewrite interpreters will make these systems very attractive.

References

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers B. V. (North-Holland), 1989.

[Bün93] R. Bündgen. Reduce the redex $\rightarrow$ ReDuX. In *Proceedings 5th Conference on Rewriting Techniques and Applications, Montreal (Canada)*, number 690 in LNCS, pages 446–450. Springer-Verlag, 1993.

[GKK+87] J. A. Goguen, Claude Kirchner, Hélène Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *LNCS*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

[Gor94] A. Gordon. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994. ISBN 0 521 47103.

[HMP95] T. Hardin, L. Maranget, and B. Pagano. Functional back-ends within the weak lambda-sigma-calculus. In *Procs. of Workshop on the Implementation of Functional Languages*, Sept. 1995.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[KKM95] C. Kirchner, H. Kirchner, and M. Vittek. Designing CLP using computational systems. In P. Van Hentenryck and S. Saraswat, editors, *Principles and Practice of Constraint Programming.* The MIT press, 1995.

[Les94] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Proceedings of the 21st Annual ACM Symposium on Principles Of Programming Languages, Portland (Or., USA)*, pages 60–69. ACM, 1994.

[LRD94] P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions $\lambda\upsilon$. Technical Report RR-2222, INRIA-Lorraine, January 1994.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, LFCS, University of Edinburgh, 1991.

[MOM93] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report CSL-93-05, SRI International, 1993.

[Vir95] P. Viry. Rewriting modulo a rewrite system. Technical Report TR-95-20, Dipartimento di Informatica, Università di Pisa, 1995.

[Vir96] P. Viry. A rewriting implementation of π-calculus. Technical Report TR-96-29, Dipartimento di Informatica, Università di Pisa, 1996.

[Vit94] Marian Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, October 1994.

Electronic Notes in Theoretical Computer Science 4 (1996)

Principles of Maude

M. Clavel [3], S. Eker [1,3], P. Lincoln [2,3], and J. Meseguer [3]

Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA

Abstract

This paper introduces the basic concepts of the rewriting logic language Maude and discusses its implementation. Maude is a wide-spectrum language supporting formal specification, rapid prototyping, and parallel programming. Maude's rewriting logic paradigm includes the functional and object-oriented paradigms as sublanguages. The fact that rewriting logic is reflective leads to novel metaprogramming capabilities that can greatly increase software reusability and adaptability. Control of the rewriting computation is achieved through internal strategy languages defined inside the logic. Maude's rewrite engine is designed with the explicit goal of being highly extensible and of supporting rapid prototyping and formal methods applications, but its semi-compilation techniques allow it to meet those goals with good performance.

1 Introduction

Maude is a logical language based on rewriting logic [17,24,20]. It is therefore related to other rewriting logic languages such as Cafe [10], ELAN [12], and DLO [6]. The equational language OBJ [11] can be regarded as a functional sublanguage of Maude.

This paper gives an introduction to the language and its interpreter implementation. Particular emphasis is placed on its basic principles and on its semantics. The style is informal, and the ideas are illustrated with simple examples to facilitate their comprehension.

The key characteristics of Maude can be summarized as follows:

[1] Supported by a NATO Fellowship administered through the Royal Society.

[2] Supported in part by Office of Naval Research Contract N00014-95-C-0168, National Science Foundation Grant CCR-9224858, and AFOSR Contract number F49620-95-C0044.

[3] Supported by Office of Naval Research Contracts N00014-95-C-022, and N00014-96-C-0114, National Science Foundation Grant CCR-9224005, DARPA through NRAD Contract N66001-95-C-8620, and by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

© *1996 Elsevier Science B. V.*

- *Based on rewriting logic.* This makes it particularly well suited to express in a declarative way concurrent and state-changing aspects of systems. Programs are theories, and rewriting logic deduction exactly corresponds to concurrent computation.
- *Wide-spectrum.* Rewriting logic is a logical and semantic framework in which specification, rapid prototyping, and efficient parallel and distributed execution, as well as formal transformations from specifications to programs can be naturally supported [14].
- *Multiparadigm.* Since rewriting logic conservatively extends equational logic [15], a equational style of functional programming is naturally supported in a sublanguage. A declarative style of concurrent object-oriented programming is also supported with a simple logical semantics. Since rewriting logic also extends Horn logic with equality in a conservative way [15], Horn logic programming can also be supported and extended in an implementation with basic facilities for unification.
- *Reflective.* Rewriting logic is reflective [8,7]. The design of Maude capitalizes on this fact to support a novel style of *metaprogramming* with very powerful module-combining and module-transforming operations that surpass those of traditional parameterized programming and can greatly advance software reusability and adaptability.
- *Internal Strategies.* The strategies controlling the rewriting process can be defined by rewrite rules and can be reasoned about inside the logic. Therefore, instead of having a "Logic+Control" introduction of extra-logical features, in Maude "Control $\subseteq$ Logic."

Maude's implementation has been designed with the explicit goals of supporting executable specification and formal methods applications, of being easily extensible, and of supporting reflective computations. Although it is an interpreter, its advanced semi-compilation techniques support flexibility and traceability without sacrificing performance. It can reach up to 200,000 rewrites per second on some applications running on a 90 MHz Sun Hyper-SPARC.

Section 2 explains the sublanguage of functional modules. An informal introduction to rewriting logic and to object-oriented modules in given in Section 3. System modules, reflection, and internal strategies are discussed in Section 4. Maude's metaprogramming capabilities are the subject of Section 5. Section 6 summarizes the semantic foundations of the language, and Section 7 describes the interpreter implementation. We conclude with some plans for the future.

2 Functional Modules

Functional modules define data types and functions on them by means of equational theories whose equations are Church-Rosser and terminating. A mathematical model of the data and the functions is provided by the *initial algebra* defined by the theory, whose elements consist of equivalence classes

of ground terms modulo the equations. Evaluation of any expression to its reduced form using the equations as rewrite rules assigns to each equivalence class a unique canonical representative. Therefore, in a more concrete way we can equivalently think of the initial algebra as consisting of those canonical representatives; that is, of the values to which the functional expressions evaluate.

As in the OBJ language [11] that Maude extends, functional modules can be unparameterized, or they can be parameterized with *functional theories* as their parameters. Functional theories have a "loose semantics," as opposed to an initial one, in the sense that any algebra satisfying the equations in the theory is an acceptable model. For example, a parameterized list module `LIST[X :: TRIV]` forms lists of models of the trivial parameter theory

```
fth TRIV is
  sort Elt .
efth
```

with one sort `Elt`; those models as just sets of elements. Similarly, a sorting module `SORTING[Y :: POSET]` sorts lists whose elements belong to a model of the `POSET` functional theory, that is, the elements must have a partial order.

The equational logic on which Maude functional modules are based is an extension of order-sorted equational logic called *membership equational logic* [16,3]; we discuss this and give more details about the semantics of functional modules in Section 6.1. For the moment, it suffices to say that, in addition to supporting sorts, subsorts, and overloading of function symbols, functional modules also support *membership axioms*, a generalization of sort constraints [23] in which a term is asserted to have a certain sort if a condition consisting of a conjunction of equations and of unconditional membership tests is satisfied.

We can illustrate these ideas with a parameterized module `PATH[G :: GRAPH]` that forms paths over a graph. This module has a path concatenation operation, has nodes as identities, and source and target functions.

```
th GRAPH is
  sorts Node Edge .
  ops s t : Edge -> Node .  *** source and target
eth

fmod PATH[G :: GRAPH] is
  sorts Path Path? .
  subsorts Node Edge < Path < Path? .
  ops s t : Path -> Node .
  op _;_ : Path? Path? -> Path? .
  var E : Edge .
  var N : Node .
  var P : Path .
  vars Q R S : Path? .
  eq (Q ; R) ; S = Q ; (R ; S) .
  cmb E ; P : Path if t(E) == s(P) .
```

```
  eq s(N) = N .
  eq t(N) = N .
  ceq s(E ; P) = s(E) if t(E) == s(P) .
  ceq t(E ; P) = t(P) if t(E) == s(P) .
  ceq N ; P = P if s(P) == N .
  ceq P ; N = P if t(P) == N .
endfm
```

Note that the concatenation of two paths is a path if and only if the target of the first is the source of the second. This follows as an inductive consequence of the simpler conditional membership axiom

```
cmb E ; P : Path if t(E) == s(P) .
```

where `E` is an edge and `P` a path. We can then instantiate this module with a concrete graph corresponding to an automaton, and can evaluate path expressions to check whether they are valid paths in the automaton.

```
fmod AUTOMATON is
  sorts Node Edge .
  ops a b c : -> Node .
  ops f g h i j : -> Edge .
  ops s t : Edge -> Node .
  eq s(f) = a .  eq t(f) = b .
  eq s(g) = c .  eq t(g) = a .
  eq s(h) = b .  eq t(h) = c .
  eq s(i) = c .  eq t(i) = b .
  eq s(j) = b .  eq t(j) = b .
endfm

make RECOGNIZER is PATH[AUTOMATON] endm
```

3 Rewriting Logic and Object-Oriented Modules

The type of rewriting typical of functional modules terminates with a single value as its outcome. In such modules, each step of rewriting is a step of *replacement of equals by equals*, until we find the equivalent, fully evaluated value. In general, however, a set of rewrite rules need not be terminating, and need not be Church-Rosser. That is, not only can we have infinite chains of rewriting, but we may also have highly divergent rewriting paths, that could never cross each by further rewriting.

The essential idea of rewriting logic [19] is that the *semantics* of rewriting can be drastically changed in a very fruitful way. We no longer interpret a term t as a functional expression, but as a *state* of a system; and we no longer interpret a rewrite rule $t \longrightarrow t'$ as an equality, but as a *local state transition*, stating that if a portion of a system's state exhibits the pattern described by t, then that portion of the system can change to the corresponding instance of t'. Furthermore, such a local state change can take place independently

from, and therefore concurrently with, any other non-overlapping local state changes. Of course, rewriting will happen *modulo* whatever structural axioms the state of the system satisfies. For example, the top level of a distributed system's state does often have the structure of a *multiset*, so that we can regard the system as composed together by an associative and commutative state constructor.

We can represent a *rewrite theory* as a four-tuple $\mathcal{R} = (\Omega, E, L, R)$, where (Ω, E) is a theory in membership equational logic, that specifies states of the system as an abstract data type, L is a set of labels, to label the rules, and R is the set of labeled rewrite rules axiomatizing the local state transitions of the system. Some of the rules in R may be conditional [19].

Rewriting logic is therefore a logic of concurrent state change. The logic's four rules of deduction—namely, reflexivity, transitivity, congruence, and replacement [19]—allow us to infer all the complex concurrent state changes that a system may exhibit, given a set of rewrite rules that describe its elementary local changes. It then becomes natural to realize that many reactive systems so specified should never terminate, and that a system may evolve in highly nondeterministic ways through paths that will never cross each other.

These ideas can be illustrated by explaining how concurrent object-oriented systems can be specified in rewriting logic, and how they can be executed using Maude's object-oriented modules.

In a concurrent object-oriented system the concurrent state, which is usually called a *configuration*, has typically the structure of a *multiset* made up of objects and messages. Therefore, we can view configurations as built up by a binary multiset union operator which we can represent with empty syntax as

```
subsorts Object Msg < Configuration .
op __ : Configuration Configuration -> Configuration
                                    [assoc comm idr: null] .
```

where the multiset union operator __ is declared to satisfy the structural laws of associativity and commutativity and to have identity *null*. The subsort declaration

```
subsorts Object Msg < Configuration .
```

states that objects and messages are singleton multiset configurations, so that more complex configurations are generated out of them by multiset union.

As a consequence, we can abstractly represent the configuration of a typical concurrent object-oriented system as an equivalence class $[t]$ modulo the structural laws of associativity and commutativity obeyed by the multiset union operator of a term expressing a union of objects and messages, i.e., as a multiset of objects and messages.

An *object* in a given state is represented as a term

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

where O is the object's name or identifier, C is its class, the a_i's are the names of the object's *attribute identifiers*, and the v_i's are the corresponding *values*.

The set of all the attribute-value pairs of an object state is formed by repeated application of the binary union operator _,_ which also obeys structural laws of associativity and commutativity; i.e., the order of the attribute-value pairs of an object is immaterial.

Consider for example a concurrent system made up of sender and receiver objects that communicate with each other by sending messages in an unreliable environment in which messages may be received out of order, some messages can be lost, and other messages can be duplicated. A fault-tolerant connection between two such objects can be accomplished by numbering the messages and sending acknowledgments back. A receiver object may have the form

```
< R : Receiver | from: S, recq: Q, reccnt: M >
```

where the attribute `from` is the name of the sending object, `recq` is the queue of received messages, and `reccnt` is the receiver's counter. In Maude, the class `Receiver` of such objects is specified by the declaration

```
class Receiver | from: OId, recq: Queue, reccnt: Nat .
```

that introduces the attribute names and the corresponding value sorts. The concurrent local state change corresponding to the reception of one message from the sender by the receiver object can then be described by the following labeled rewrite rule.

```
rl [ receive ] :
   < R : Receiver | from: S, recq: Q, reccnt: M >
   (to: R (E,N))
=> < R : Receiver | from: S,
                    recq: (if N == s(M) then push(Q,E) else Q fi),
                    reccnt: (if N == s(M) then s(M) else M fi) >
   (to: S ack N) .
```

That is, the new value `E` is appended to the queue and the counter is increased iff the number `N` in the message is `M + 1`; otherwise, the message is discarded and the receiver does not change its state, but in any case an acknowledgment is always sent to the sender.

The entire fault-tolerant protocol for sender and receiver objects—discussed in a somewhat different way in Chandy and Misra [5], and similar in some ways to the presentation of the alternating bit protocol by Lam and Shankar [13]—can be defined in the following parameterized object-oriented module.

Note that Maude's syntax for object-oriented modules leaves implicit some well-understood assumptions, such as the syntax for objects, the existence of a multiset union operator to form configurations, and the conventions for class inheritance. However, object-oriented modules can be systematically translated into ordinary rewrite theories by making explicit all these assumptions. They can therefore be understood as a special case of system modules. A detailed account of this translation process can be found in [20].

```
omod PROTOCOL[ELT :: TRIV] is
  protecting QUEUE[ELT] .
```

```
sort Contents Count .
subsort Elt < Contents .
op z : -> Count .
op s_ : Count -> Count .
op empty : -> Contents .
msg to:_(_,_) : OId Elt Count -> Msg . *** data to receiver
msg to:_ack_ : OId Count -> Msg . *** acknowledgment to sender
class Sender | rec: OId, sendq: Queue, sendbuff: Contents,
               sendcnt: Count, repcount: Count .
class Receiver | from: OId, recq: Queue, reccnt: Count .
vars S R : OId .
vars N M X : Count .
var E : Elt .
var Q : Queue .
var C : Contents .

rl [ produce ] :
   < S : Sender | rec: R, sendq: cons(E, Q), sendbuff: empty,
                  sendcnt: N, repcount: X > =>
   < S : Sender | rec: R, sendq: Q, sendbuff: E,
                  sendcnt: s(N), repcount: s(s(s(z))) > .

rl [ send ] :
   < S : Sender | rec: R, sendq: Q, sendbuff: E,
                  sendcnt: N, repcount: s(X) > =>
   < S : Sender | rec: R, sendq: Q, sendbuff: E,
                  sendcnt: N, repcount: X >
   (to: R (E,N)) .

rl [ rec-ack ] :
   < S : Sender | rec: R, sendq: Q, sendbuff: C,
                  sendcnt: N, repcount: X >
   (to: S ack M) =>
   < S : Sender | rec: R, sendq: Q,
                  sendbuff: (if N == M then empty else C fi),
                  sendcnt: N, repcount: X > .

rl [ receive ] :
   < R : Receiver | from: S, recq: Q, reccnt: M >
   (to: R (E,N))
=> < R : Receiver | from: S,
                    recq: (if N == s(M) then push(Q,E) else Q fi),
                    reccnt: (if N == s(M) then s(M) else M fi) >
   (to: S ack N) .
endom
```

These definitions will generate a reliable, in-order communication mechanism

from an unreliable one. The message counts are used to ignore all out-of-order messages, and the replication count is used to replicate messages that may be lost if the channel is faulty. The fairness assumptions of Maude will ensure that the `send` action and corresponding `receive` actions will be repeated until a `rec-ack` can be performed, or the replication counter goes to zero. One can directly represent unbounded retransmission by eliminating this check as well, although the protcol then relies more strongly on fairness assumption. In [24,20] it is explained how we can also model some fault modes of the communication channel by additional rewrite rules which duplicate or destroy messages declared in a module extending the one above.

Formally, letting C denote the initial configuration of objects and C' denote configuration resulting after rewriting, we have been able to deduce the sentence $C \longrightarrow C'$ as a logical consequence of the rewrite rules in the module. Indeed, the rules of deduction of rewriting logic support sound and complete reasoning about the concurrent transitions that are possible in a concurrent system whose basic local transitions are axiomatized by given rewrite rules. That is, the sentence $[t] \longrightarrow [t']$ is provable in the logic using the rewrite rules that axiomatize the system as axioms if and only if the concurrent transition $[t] \longrightarrow [t']$ is possible in the system.

In this object-oriented case we make several implicit assumptions, including the associativity and commutativity of the multiset union operator. In general system modules, however, the axioms E can be varied as a very flexible parameter to specify many different types of concurrent systems. In this way, rewriting logic can be regarded as a very general semantic framework for concurrency that encompasses a very wide range of well-known models [19,22].

Maude's default interpreter can be quite adequate for simulating concurrent object-oriented systems. However, for the purposes of studying a system in depth—for example, by exploring all the possible rewrites from a given state to another—or of controlling the possibly highly nondeterministic evolution of a system that need not be object-oriented, we need other means.

4 System Modules, Strategies, and Reflection

The most general Maude modules are system modules. They specify the initial model of a rewrite theory $\mathcal{R}$ [19]. This initial model is a transition system whose states are equivalence classes $[t]$ of ground terms modulo the equations E in $\mathcal{R}$, and whose transitions are proofs $\alpha : [t] \longrightarrow [t']$ in rewriting logic—that is, concurrent computations in the system so described. Such proofs are equated modulo a natural notion of proof equivalence that computationally corresponds to the "true concurrency" of the computations.

Consider for example a system module `NIM` specifying a version of the game of Nim. There are two players and two bags of pebbles: a "draw" bag to remove pebbles from, and a "limit" bag to limit the number of pebbles that can be removed. The two players take turns making moves in the game. At each move a player draws a nonempty set of pebbles not exceeding those in the limit bag. The limit bag is then readjusted to contain the least number

of pebbles in either the double of what the player just drew, or what was left in the draw bag. The game then continues with the two bags in this new state. The player who empties the draw bag wins. An intermediate move is axiomatized by the rule `[mv]`; the last, winning move is axiomatized by the rule `win`.

```
mod NIM is
  protecting BOOL .
  sorts Pebble Bag State .
  subsorts Pebble < Bag .
  op o : -> Pebble .
  op nil : -> Bag .
  op __ : Bag Bag -> Bag [assoc comm] .
  op _=<_ : Bag Bag -> Bool .
  op least : Bag Bag -> Bag .
  op state : Bag Bag -> State .
  vars X Y Z : Bag .
  eq o nil = o .
  eq nil =< X = true .
  eq o X =< nil = false .
  eq o =< o  = true .
  eq o =< o X = true .
  ceq o X  =< o = false if X =/= nil .
  eq o X =< o Y = X =< Y .
  eq least(X,Y) = if X =< Y then X else Y fi .
  crl [mv] : state(X Y,Z) => state(Y,least(X X,Y))
                             if X =< Z and X =/= nil .
  crl [win] : state(X,Y) => state(nil,nil)
                            if X =< Y and X =/= nil .
endm
```

The initial model described by this module is the transition system containing exactly all the possible game moves allowed by the game. But there are many bad moves that would allow the other player to win. A good player should avoid such bad moves by having a *winning strategy*. With such a strategy, each move made by the player inexorably leads to success, no matter what moves the other player attempts.

What we obviously want, in this and in many other examples, is to have good ways of controlling the rewriting inference process—which in principle could go in many undesired directions—by means of adequate *strategies*. Many systems, for example theorem provers and declarative languages implementations, support certain strategies of this nature. However, such strategies are often *external* to the languages they control: they may constitute a separate programming language external to the logic, or may be part of the language's "extralogical features." In Maude, thanks to the reflective capabilities of rewriting logic, strategies can be made *internal* to rewriting logic. That is, they can be defined by rewrite rules, and can be reasoned about as with rules

in any other theory. The value of specifying strategies with rewrite rules is also emphasized in the most recent work on ELAN [2].

In fact, there is great freedom for defining many different strategy languages inside Maude. This can be done in a completely user-definable way, so that users are not limited by a fixed and closed strategy language. Also, even if some users decide to adopt a particular strategy language because of its good features, such a language remains fully *extensible*, so that new features and new strategy concepts can be defined on top of them. Of course, such languages should be defined in a disciplined way that guarantees that they are correct, that is, that they only produce valid rewrites, as we explain below.

In Maude, a strategy language is a *function on theories*, that assigns to a module M another module $strat(M)$, whose terms are called *strategy expressions* specifying desired, possibly quite complex, set of rewrite deductions in the original theory M. Executing such a strategy expression is simply rewriting it using the rules in $strat(M)$. In some cases, such executions may never terminate. However, as the expression is being rewritten, more and more of the desired rewrites in the theory M that the strategy expression in question was supposed to describe become directly "visible" in the partially rewritten strategy expression. In this way, we can tame the wildness of M by shifting our ground to a much more controllable theory $strat(M)$. For example, $strat(M)$ may be Church Rosser, and therefore essentially a functional module, so that computations of strategy expressions become essentially deterministic. This is of course not a necessary requirement, but it is nevertheless an attractive possibility in the context of a sequential implementation.

We first briefly discuss reflection in rewriting logic and then explain how it can be used to define and give semantics to internal strategy languages. Rewriting logic is reflective [8,7]. That is, there is a rewrite theory $\mathcal{U}$ with a finite number of operations and rules that can simulate any other finitely presentable rewrite theory $\mathcal{R}$ in the following sense: given any two terms t, t' in $\mathcal{R}$ there are corresponding terms $\langle \overline{\mathcal{R}}, \overline{t} \rangle$ and $\langle \overline{\mathcal{R}}, \overline{t'} \rangle$ in $\mathcal{U}$ such that we have

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

Let us denote by $FPTh$ the class of finitely presented rewrite theories. An *internal strategy language* is a theory-transforming function $strat : FPTh \longrightarrow FPTh$ that satisfies specific semantic requirements [8,7]. A sound methodology for defining such languages is to first define a *strategy language kernel* as a function, say, $meta : FPTh \longrightarrow FPTh$ that sends $\mathcal{R}$ to a definitional extension of $\mathcal{U}$—or a suitable subtheory of $\mathcal{U}$—by rewrite rules defining how rewriting in $\mathcal{R}$ is accomplished at the metalevel. A typical semantic definition that one wants to have in $meta(\mathcal{R})$ is that of $metaapply(\overline{l}, \overline{t})$, that simulates at the metalevel one step of rewriting at the top of a term t using the rule labeled `l` in $\mathcal{R}$. Proving the correctness of such a small strategy language kernel is then quite easy, by using the correctness of $\mathcal{U}$ itself as a universal theory. The next step is to define a strategy language of choice, say *strat*, as a function sending each theory $\mathcal{R}$ to a theory that extends $meta(\mathcal{R})$ by additional strategy expressions and corresponding semantic rules, all of which are recursive definitional extensions of those in the kernel in an appropriate

sense, so that their correctness can then be reduced to that of the kernel.

The descriptions of *meta* and *strat* that we have just given are phrased in metalevel terms, that is, they are described as metalevel functions. But in fact they are definable as functions within rewriting logic. Note that in $\mathcal{U}$ the theory $\mathcal{R}$ is represented as a *term* $\overline{\mathcal{R}}$. In fact, assuming a sorted version of the logic, all such terms $\overline{\mathcal{R}}$ are the elements of a sort `Module` in $\mathcal{U}$. This means that any effective function $F : FPTh \longrightarrow FPTh$ mapping a finitely presentable rewrite theory to another at the metalevel of the logic can now be represented at the object level as a computable function $\overline{F}$: `Module` $\longrightarrow$ `Module`. Therefore, by the metatheorem of Bergstra and Tucker [1], we can always specify such a function by a finite set of Church-Rosser and terminating rewrite equations in a suitable conservative extension of $\mathcal{U}$.

More details on the semantic definition of an internal strategy language for a logic in general, and for rewriting logic in particular, can be found in [7]. Since the rewrite engine can be naturally regarded as an implementation of key functionality in the universal theory $\mathcal{U}$, the Maude implementation supports a strategy kernel `META<X : Module>` in a built-in fashion for greater efficiency. The definition of a concrete strategy language `STRAT` as a functional module extending `META` is given in Appendix 9.

A strategy expression in `STRAT` initially has the form

```
rew T => ? with S
```

where `T` stands for the representation $\bar{t}$ in $\mathcal{U}$ of a term t in the object theory $\mathcal{R}$ in question—for example, the two pebble bag (`o o`) in `NIM` has the representation `'_[_]['o,'o]` in `STRAT<NIM>`—and `S` is the rewriting strategy that we wish to compute. The symbol `?` indicates that we are beginning the computation of such a strategy; as the computation proceeds, `?` gets rewritten into a *tree of solutions*, and `S` is rewritten into the remaining strategy to be computed. In case of termination, this is the `idle` strategy and we are done.

This language can then be used to find a winning strategy for the `NIM` example. Such a strategy can easily be defined by extending the basic module `STRAT<NIM>` with a couple of mutually recursive strategies `movetowin` and `findawinner`

```
fmod NIM-WIN is
extending STRAT <NIM> .
ops mv win : -> Label .
ops movetowin findawinner : -> StrategyName .
vars T T' : Term . var S1T : SolTree . var S1TL : SolTreeList .

eq rew T => S1T{<- T'} with movetowin =
   rew T => S1T{<- T'} with
       (apply(win);; idle
        orelse (dk-apply(mv); findawinner)) .

eq rew T => S1T{<- mk(S1TL)} with findawinner =
   rew T => S1T{<- mk(S1TL)}
```

```
    with downleft ; (movetowin ;; (prunesol ; findawinner)
                     orelse (prunerest ; up)) .

endfm
```

Intuitively, given a state $\langle X, Y \rangle$ in the game, `movetowin` will find a *winning move* $\langle X', Y' \rangle$ for a player A if there is one, in the sense that either $\langle X', Y' \rangle = \langle nil, nil \rangle$ or $\langle X', Y' \rangle$ is a move that eventually will lead the player A to success, no matter what moves the player B attempts, assuming that in the following moves, the player A always plays with the strategy `movetowin`.

In particular, `movetowin` defines the following strategy for a player A given a state $\langle X, Y \rangle$ in the game: try to win the game with just one move (`apply(win)`); if not, create a tree whose leaves $\langle X'_i, Y'_i \rangle$, $1 \leq i \leq n$, are all the allowed moves from the state $\langle X, Y \rangle$ (`dk-apply(mv)`). Then, try to find a leaf $\langle X'_i, Y'_i \rangle$ representing a state from which the player B can not make a winning move (`findawinner`); if not, the result of the strategy `movetowin` for the player A will be `failure`.

As expected, `findawinner` defines the following strategy for a player A over a tree T (possibly empty) of allowed moves: try to select the first leaf $\langle X'_1, Y'_1 \rangle$ of T (`downleft`); note that if T is empty, the result of `downleft` will be `failure`. Then, if the player B can make a winning move from $\langle X'_1, Y'_1 \rangle$ (`movetowin`), prune that leaf (`prunesol`) and try to find among the rest of the leaves a winning move (`findawinner`); if the player B can not make a winning move from $\langle X'_1, Y'_1 \rangle$, prune the rest of the leaves (`prunerest`) and select $\langle X'_1, Y'_1 \rangle$ (`up`) as a winning move.

We can then run the following examples to find a winning move when there is one, or to fail to do so otherwise.

```
Maude>red rew 'state[('__['o,'o,'o,'o]),('__['o,'o,'o])] => ?
              with movetowin .
Result in sort StrategyExp:
      rew 'state[('__['o,'o,'o,'o]),('__['o,'o,'o])] =>
           ^{<- 'state['__['o,'o,'o],'__['o,'o]]} with idle .
Maude>red rew 'state[('__['o,'o,'o,'o,'o]),('__['o,'o,'o,'o])]
               => ? with movetowin .
Result in sort StrategyExp: failure .
```

5 Metaprogramming in Maude

Perhaps one of the most important new contributions of Maude is the *metaprogramming methodology* that it supports in a simple and powerful way. This methodology is well integrated with the language's semantic foundations, particularly with its logical foundations for reflection.

By "metaprogramming" we of course mean the capacity of defining programs that operate on other programs as their data; in our case, equational and rewrite theories that operate on other such theories as their data. By observing that we can not only reify theories, but also views among them, this

includes the more traditional "parameterized programming" capabilities in the Clear-OBJ tradition [4,11] as a particular instance. The difference is that in that tradition theories are metalevel entities not accessible at the object level of the logic, since this is only possible in an explicitly reflective logical context.

What reflection accomplishes is to *open up to the user* the metalelevel of the language, so that instead of having a fixed repertoire of parameterized programming operations we can now define a much wider range of theory-transforming and theory-combining operations that could not be defined using more traditional means. We have illustrated this power with the $meta(X : \texttt{Module})$ and $strat(X : \texttt{Module})$ constructions, that are "parameterized modules" in this much more general sense. Another good example, given in [15], is the reification of the logic map $\Psi : LLogic \longrightarrow RWLogic$ from linear logic to rewriting logic as an equationally defined function $\overline{\Psi} : \texttt{LLTheory} \longrightarrow \texttt{Module}$ inside rewriting logic. This example illustrates a general method by which, when using rewriting logic as a logical framework, we can always reify an effectively given map of logics $\Phi : \mathcal{L} \longrightarrow RWLogic$, sending finitely presentable theories in $\mathcal{L}$ to finitely presentable rewrite theories, as an equationally defined function $\overline{\Psi} : \texttt{Theory}_{\mathcal{L}} \longrightarrow \texttt{Module}$ inside rewriting logic.

Many more examples could be given. Indeed, we plan to systematically exploit Maude's metaprogramming capabilities to make the language and its environment very easily extensible and modifiable, and to support many logical framework and semantic framework applications such as: representation and interoperation of logics inside rewriting logic, executable definition of other logical languages in Maude, and definition of theorem-proving environments and tools for Maude and for other languages inside rewriting logic.

In summary, what reflection makes possible in Maude is the definition of an open, extensible, and user-definable *module algebra* supporting a new style of metaprogramming with very promising advantages for software methodology.

6 The Semantics of Maude

We summarize the semantic foundations of Maude's functional, object-oriented, and system modules.

6.1 Membership equational logic and functional modules

Maude is a declarative language based on rewriting logic. But rewriting logic has its underlying equational logic as a parameter. There are for example unsorted, many-sorted, and order-sorted versions of rewriting logic, each containing the previous version as a special case. The underlying equational logic chosen for Maude is *membership equational logic* [16,3], a conservative extension of both order-sorted equational logic and partial equational logic with existence equations [16]. It supports partiality, subsorts, operator overloading, and error specification.

A *signature* in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ with

K a set of *kinds*, (K, Σ) a many-sorted (although it is better to say "many-kinded") signature, and $S = \{S_k\}$ a K-kinded set of *sorts*.

An *Ω-algebra* is then a (K, Σ)-algebra A together with the assignment to each sort $s \in S_k$ of a subset $A_s \subseteq A_k$. Intuitively, the elements in sorts are the good, or correct, or nonerror, or defined, elements, whereas the elements without a sort are error or undefined elements.

Atomic formulas are either Σ-equations, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. General sentences are Horn clauses on these atomic formulae, quantified by finite sets of K-kinded variables. That is, they are either conditional equations

$$(\forall X)\ t = t\ \ if\ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

or *membership axioms* of the form

$$(\forall X)\ t : s\ \ if\ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j).$$

Membership equational logic has all the usual good properties: soundness and completeness of appropriate rules of deduction, initial and free algebras, relatively free algebras along theory morphisms, and so on [16].

In Maude, *functional modules* are equational theories in membership equational logic satisfying additional requirements. The semantics of an unparameterized functional module is the initial algebra specified by its theory; the semantics of a parameterized functional module is the free functor associated to the inclusion of the parameter theory. *Functional theories* are also membership equational logic theories, but they have instead a loose interpretation, in that all models of the theory are acceptable, although a functional theory may impose the additional requirement that some of its subtheories should be interpreted initially. This is entirely similar to the treatment of "objects" and theories in OBJ [11]. Indeed, since membership equational logic conservatively extends order-sorted equational logic, Maude's functional modules extend OBJ modules.

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. There is no need to declare kinds explicitly. The convenience of order-sorted notation is retained as syntactic sugar. Thus, an operator declaration

```
op push : Nat Stack -> NeStack .
```

is understood as the membership axiom

$$(\forall x, y)\ push(x, y) : NeStack\ \ if\ \ x : Nat \wedge y : Stack.$$

Similarly, a subsort declaration `NeStack < Stack` corresponds to the membership axiom

$$(\forall x)\ x : Stack\ \ if\ \ x : NeStack.$$

Computation in a functional module is accomplished by using the equations as rewrite rules until a canonical form is found. Therefore, the equations must satisfy the additional requirements of being Church-Rosser, terminating, and sort-decreasing [3]. This guarantees that all terms in an equivalence

class modulo the equations will rewrite to a unique canonical form, and that this canonical form can be assigned a sort that is smaller than all other sorts assignable to terms in the class. For a module satisfying such conditions any reduction strategy will reach a normal form; nevertheless, the user can assign to each operator a functional evaluation strategy in the OBJ style [11] to control the reduction for efficiency purposes. If no such strategies are declared, a bottom-up strategy is chosen. Since Maude supports rewriting modulo equational theories such as associativity or associativity/commutativity, all that we say has to be understood for equational rewriting *modulo* such axioms.

In membership equational logic the Church-Rosser property of terminating and sort-decreasing equations is indeed equivalent to the confluence of their critical pairs [3]. Furthermore, both equality and membership of a term in a sort are then *decidable* properties [3]. That is, the equality and membership predicates are *computable functions*. We can then use the metatheorem of Bergstra and Tucker [1] to conclude that such predicates are themselves specifiable by Church-Rosser and terminating equations as Boolean-valued functions. This has the pleasant consequence of allowing us to include inequalities $t \neq t'$ and negations of memberships $not(t : s)$ in conditions of equations and of membership axioms, since such seemingly negative predicates can also be axiomatized *inside the logic* in a positive way, provided that we have a subspecification of (not necessarily free) constructors in which to do it, and that the specification is indeed Curch-Rosser, terminating, and sort decreasing. Of course, in practice they do *not* have to be explicitly axiomatized, since they are built into the implementation of rewriting deduction in a much more efficient way.

Let us denote membership equational logic by $Eqtl^:$ and its associated rewriting logic by $RWLogic^:$. Regarding an equational theory as a rewrite theory whose set of rules is empty defines a conservative map of logics [15]

$$Eqtl^: \longrightarrow RWLogic^:$$

This is the way in which Maude's functional modules are regarded as a special case of its more general system modules.

6.2 Semantics of object-oriented and system modules

As already pointed out, the logic of Maude is the membership logic variant of rewriting logic $RWLogic^:$. A *system module* is then a rewrite theory. In the unparameterized case its semantics is the initial model defined by the theory [19], which is the algebra of all rewriting computations for ground terms in the theory. From a systems perspective this model describes all the concurrent behaviors that the system so axiomatized can exhibit. From that perspective a term t denotes a state of the system, and a rewrite $t \longrightarrow t'$ denotes a possibly concurrent computation.

A system module can contain one or more parameter theories. The inclusion from the parameter(s) into the module then gives rise to a free extension functor [18], which provides the semantics for the module. This of course means that we can compose systems by putting together the rewrite theories

in which they are specified.

A rewrite theory has both rules and equations, so that rewriting is performed *modulo* such equations. However, this does not mean the Maude implementation must have a matching algorithm for each equational theory that a user might specify, which is impossible, since matching modulo an arbitrary theory is undecidable. What we instead require for theories in system modules is that:

- the equations are divided into a set A of axioms, for which matching algorithms exist in the Maude implementation [4], and a set E of equations that are Church-Rosser, terminating and sort decreasing *modulo* A; that is, the equational part must be equivalent to a functional module;
- The rules R in the module are *coherent* [26] (or at least what might be called "weakly coherent" [20], Section 5.2.1) with the equations E modulo A. This means that appropriate critical pairs exist between rules and equations allowing us to intermix rewriting with rules and rewriting with equations in any way without losing rewrite computations by failing to perform a rewrite that would have been possible before an equational deduction step was taken. In this way, we get the effect of rewriting modulo $E \cup A$ with just a matching algorithm for A. In particular, a simple strategy available in these circumstances is to always reduce to canonical form using E before applying any rule in R.

Since the state of the system specified by a system module is axiomatized as an abstract data type by the equations E modulo A, and the rules in R are local rules for changing such a state, in practice the lefthand sides of rules in R only involve constructor patterns, so that coherence is a natural byproduct of good specification practice. Besides, using the completion methods in [26] one can check coherence, and one can try to make a set of rules coherent when they are not so.

The semantics of object-oriented modules is entirely reducible to that of system modules, in the sense that there is a systematic desugaring process translating each object-oriented module into its corresponding system module. However, the particular ontology supported by object-oriented modules is something very much worth keeping, and it does not exist for general system modules. For example, in an object-oriented configuration we have objects that maintain their *identity* across their state changes, and the notions of fairness adequate for them are more specialized than those appropriate for arbitrary system modules. The approach taken in Maude is to provide a logical semantics for concurrent object-oriented programming by taking rewriting logic as its foundation, and then defining in a rigorous way higher-level object-oriented concepts above such a foundation. The papers [20,21] provide good background on such foundations. Talcott's paper [25] gives rewriting logic

[4] Maude's rewrite engine has an extensible design, so that matching algorithms for new theories can be added and can be combined with existing ones [9]. At present, matching modulo associativity and commutativity, and a preliminary version of matching modulo associativity are supported.

foundations for actors from a somewhat different viewpoint.

The basic ideas about the reflective semantics of Maude have already been discussed in Section 4. Much more detail can be found in [7].

7 The Maude Implementation

This section describes the implementation of the Maude interpreter, which consists of two main components: the front end and the engine.

7.1 Front end and module evaluation

The front end of the Maude interpreter is built on top of the OBJ3 front end, and is written in Common Lisp. The Maude front end shares with OBJ3 the convenient mixfix syntax for user-defined symbols and expressive parameterized programming mechanisms. The Maude front end augments this with additional syntax for Maude language constructs, tracing and debugging commands, complete disambiguation of ad-hoc overloaded operators, a complete module-flattening operation, a specialized pretty- and unpretty-printer, a program transformation from object-oriented modules to system modules, and support for meta-level specifications. The result is that users can enter Maude specifications using powerful parameterized programming constructs and mixfix syntax which are completely eliminated before a Maude specification is passed to the engine. Output from the engine is passed back through a pretty-printer which reparses the output in prefix form, and then prints the result in the user-declared mixfix style. Timing and rewriting statistics from the engine are also reported from the engine to the user through the front end.

7.2 Maude's rewrite engine

The design objectives of the Maude rewrite engine are consistent with the executable specification and formal method uses that we wish to support. The system should "look and feel" like an interpreter, should be capable of supporting user interrupts and source level tracing, and above all should be extensible with new equational theories and new built-in operators both of which may require new term/data representations to be integrated seamlessly with existing term/data representations. Reflective capabilities are also central to our design, since the system should support arbitrary levels of meta-rewriting.

Although we have sought the most efficient implementation meeting the above objectives, supporting them all but rules out a number of performance enhancing techniques such as: compilation to native machine code (or C); compilation to a fixed architecture abstract machine; program transformations and partial evaluation; and tight coupling between the matching/ replacement/ normalization code for different equational theories—i.e., where code operating on symbols in one equational theory recognizes symbols in alien theories and makes use of their properties.

The design chosen is essentially a highly modular semi-compiler where the most time consuming run-time tasks are compiled at parse-time into a sys-

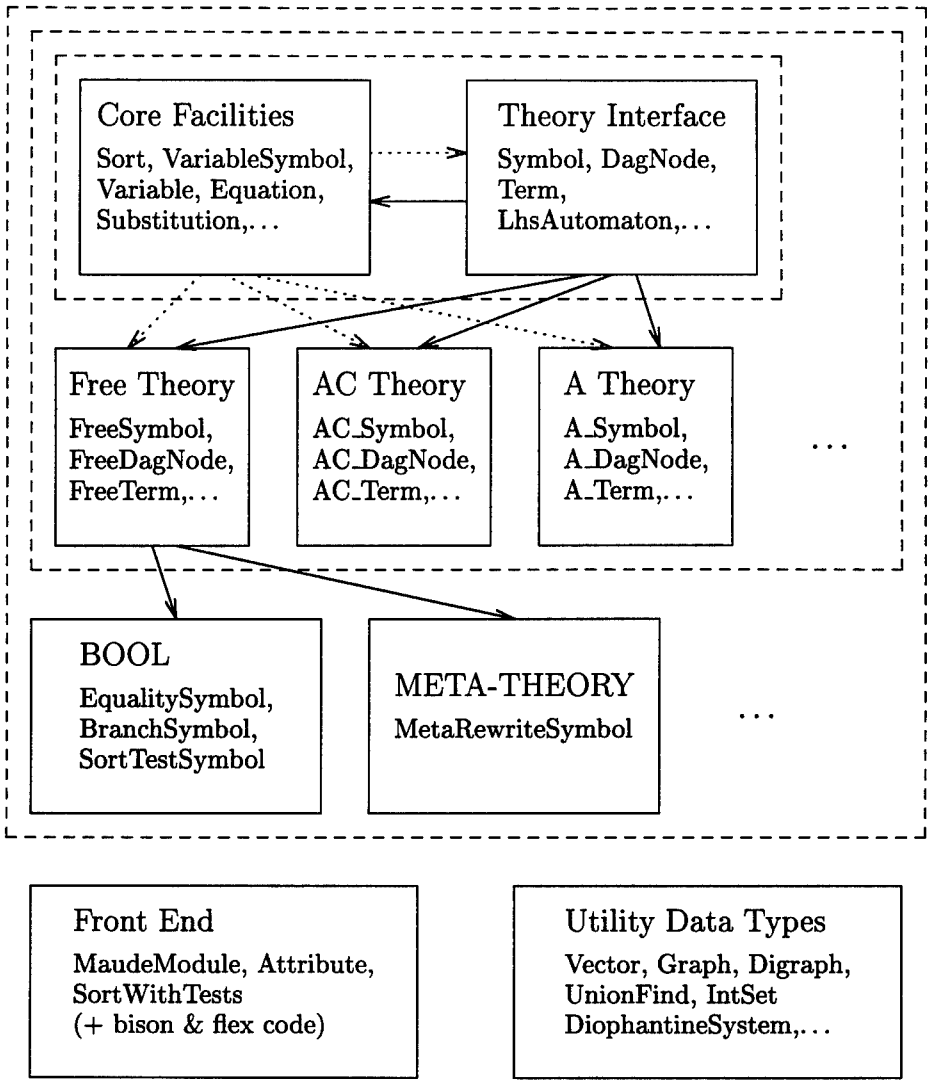


Fig. 1. Overall structure of the Maude Interpreter's Rewrite Engine

tem of lookup tables and automata which are interpreted at run-time. After some early experiments it was found very useful to have two distinct representations for terms. For most uses terms are represented as trees, in which nodes are decorated with all kinds of information to simplify parse time analysis. For the subject term being rewritten, however, a directed-acyclic-graph (DAG) representation is used with very compact nodes. Heavy use is made of object-oriented structuring techniques and great care has been taken to ensure extensibility and to make the bulk of the engine application-independent.

The overall structure of the rewrite engine is shown in Figure 1, where each module is shown as a box and some of the names of the modules classes are shown in each box.

Solid arrows indicate that some of the classes in the target module are derived from classes in the source module; dotted arrows indicate that classes in the target module use facilities provided by the source module. The modules themselves are organized in a layered structure where inner layers have no knowledge of, or dependency on, outer layers.

The innermost layer consists of the modules *Core Facilities* and *Theory Interface.* The Theory Interface consists of abstract classes for basic objects whose concrete realization will differ for different equational theories, such as: symbols, dag nodes, terms, lefthand side automata (for matching), righthand side automata (for constructing and normalizing righthand side and condition instances), matching subproblems and matching extension information. Some of the classes in the Theory Interface contain some concrete data and function members to provide useful common functionality to derived classes. The Core Facilities consists of concrete classes for basic objects that are independent of the different equational theories, such as: sorts, connected components (kinds), variable symbols, variables (as terms), equations, sort constraints, rules, sequences of matching subproblems and substitutions. Neither the Core Facilities nor the Theory Interface treat any sort, symbol or equational theory as special in any way whatsoever; all are manipulated through virtual functions in the abstract classes belonging to the Theory Interface. In particular, this means that the code that handles conditional equations knows nothing about the Maude built in sort *Bool* and its built in constants *true* and *false.* Instead conditional equations always have the form

$$l = r \text{ if } c_1 = c_2.$$

and if a more complex boolean condition b is desired, it is encoded as the equality $b = \mathit{true}$.

The next layer consists of modules for individual equational theories. Each module in this layer consists of concrete descendents of abstract classes from the Theory Interface, which provide a theory specific implementation of virtual functions such as *match*(), *compileLhs*() and *rewrite*(). In this way each equational theory has its own representation objects such as symbols, terms, dag nodes and matching automata. At this level there are no special sorts or symbols and each module is only aware of the representation of its own classes; everything else is *alien* and is manipulated through the Theory Interface.

The next layer consists of modules containing classes which provide symbols with non-standard run-time properties. Even here there are no special sorts or symbols; only classes for symbols that have rather generalized non-standard run-time behavior. The *BranchSymbol* class for example can be used to generate all manners of conditional constructs including the 'if-then-else-fi' needed for Maude. These classes only affect the behaviour of a symbol when an attempt is made to rewrite at a dag node containing it. All other properties (such as matching and normalization) and data representations are inherited from the parent equational theory.

The outermost module *Front End* contains a rudimentary parser, the class *MaudeModule* and a couple of minor classes. Only here do Maude specific operators such as 'if-then-else-fi' and 'meta-apply' really exist. The Front End is dependent on all the other modules but no other module depends on it. It can be changed or replaced without modifying the rest of the engine.

One final module is the *Utility Data Types.* This contains classes and class templates implementing 'components of general utility' such as vectors, graphs

and Tarjan's union-find data structure. These are used freely throughout the engine.

Performance enhancing techniques implemented in the current prototype include:

(i) Fixed size dag nodes for in-place replacement.

(ii) Full indexing for the topmost free function symbol layer of patterns; when the patterns for some free symbol only contain free symbols this is equivalent to matching a subject against all the patterns simultaneously.

(iii) Use of *greedy matching algorithms*, which attempt to generate a single matching substitution as fast as possible for patterns and subpatterns that are simple enough and whose variables satisfy certain conditions (such as not appearing in a condition). If a greedy matching algorithm fails it may be able to report that no match exists; but it is also allowed to report 'undecided' in which case the full matching algorithm must be used.

(iv) Use of binary search during AC matching for fast elimination of ground terms and previously bound variables.

(v) Use of a specially designed sorting algorithm which uses additional information to speed up the renormalization of AC terms.

(vi) Use of a Boyer-Moore style algorithm for matching under associative function symbols.

(vii) Compile time analysis of sort information to avoid needless searching during associative and AC matching.

(viii) Compile time analysis of non-linear variables in patterns in order to propagate constraints on those variables in an 'optimal' way and reduce the search space.

(ix) Compile time allocation of fixed size data structures needed at run time.

(x) Caching dynamically sized data structures created at run time for later reuse if they are big enough.

(xi) Bit vector encoding of sort information for fast sort comparisons.

(xii) Compilation of sort information into *regularity tables* for fast incremental computation of sorts at run time.

(xiii) Efficient handling of *matching with extension* through a theory independent mechanism that avoids the need to extension variables or equations.

In large examples involving the free theory, we have observed speedups in the order of 35–55 times faster than the OBJ3 implementation, reaching up to 200,000 rewrites per second on a 90 MHz Sun HyperSPARC. For examples of associative commutative rewriting we have observed typical speeds of 10,000 rewrites per second, and in some cases three or more orders of magnitude speedup over OBJ3.

The current version of the engine comprises 79 classes implemented by approximately 19500 lines of C++.

8 Future Plans

We have introduced the main ideas and the basic principles of Maude and have illustrated them with examples. In addition to continued work on theoretical foundations much more experimentation and implementation work lies ahead of us. The following areas will receive special attention:

- Further development of, and experimentation with, Maude's reflective and metaprogramming capabilities.
- Experimentation with different strategy languages, development of useful strategy libraries, and study of parallel strategies.
- Extension of the rewrite engine with matching algorithms for new equational theories.
- Implementation of unification algorithms to support narrowing computations in addition to rewriting. This will also allow adequate treatment of rules with extra variables in their righthand sides, that are not supported by the current implementation.
- Development of a theorem-proving environment supporting automated reasoning about specifications in Maude and in other languages.
- Implementation of foreign interface modules [24,20], to support frequently occurring computations in a more efficient, built-in way.
- Input-Output. This should be naturally specified using Maude's concurrent object-oriented concepts.
- Compilation of Maude, as well as parallel and distributed implementations of the language.
- Applications and case studies. Application areas that seem particularly promising include: logical framework applications, module algebra and metaprogramming methodology, object-oriented applications, symbolic simulation, real-time system specification, parallel programming, and uses of Maude as a programming language definition and prototyping tool.

Acknowledgements
We cordially thank Timothy Winkler and Narciso Martí-Oliet for their valuable contributions to the development of the Maude ideas. We also thank Carolyn Talcott for many discussions on Maude and for her valuable suggestions on strategy aspects. We are grateful for very helpful discussions and exchanges with Kokichi Futatsugi, Claude and Hélène Kirchner, Martin Wirsing, Ulrike Lechner, Christian Lengauer, and many other colleagues. Our previous work with Joseph Goguen and the other members of the OBJ team has also influenced the development of our ideas.

References

[1] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van

Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90. Springer-Verlag, 1980. LNCS, Volume 81.

[2] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. This volume.

[3] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. Manuscript, SRI International, August 1996.

[4] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer LNCS 86, 1980.

[5] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[6] A. Ciampolini, E. Lamma, P. Mello, and C. Stefanelli. Distributed logic objects: a fragment of rewriting logic and its implementation. This volume.

[7] Manuel G. Clavel and José Meseguer. Reflection and strategies in rewriting logic. This volume.

[8] Manuel G. Clavel and José Meseguer. Axiomatizing reflective logics and languages. In Gregor Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.

[9] Steven Eker. Fast matching in combination of regular equational theories. This volume.

[10] K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Proc. of the Kunming International CASE Symposium, Kunming, China, November*, 1994.

[11] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1996. To appear in J.A. Goguen, editor, *Applications of Algebraic Specification Using OBJ*, Cambridge University Press.

[12] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In V. Saraswat and P. van Hentryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 133–160. MIT Press, 1995.

[13] Simon S. Lam and A. Udaya Shankar. A relational notation for state transition systems. *IEEE Transactions on Software Engineering*, SE-16(7):755–775, July 1990.

[14] Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G.E. Blelloch, K.M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, pages 309–339. DIMACS Series, Vol. 18, American Mathematical Society, 1994.

[15] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[16] José Meseguer. Membership algebra. Lecture at the Dagstuhl Seminar on "Specification and Semantics," July 9, 1996. Extended version in preparation.

[17] José Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.

[18] José Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02, SRI International, Computer Science Laboratory, February 1990. Revised June 1990.

[19] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[20] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[21] José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In Oscar M. Nierstrasz, editor, *Proc. ECOOP'93*, pages 220–246. Springer LNCS 707, 1993.

[22] José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proceedings of the CONCUR'96 Conference, Pisa, August 1996*. Springer LNCS, 1996.

[23] José Meseguer and Joseph Goguen. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. *Information and Computation*, 103(1):114–158, 1993.

[24] José Meseguer and Timothy Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, pages 253–293. Springer LNCS 574, 1992. Also Technical Report SRI-CSL-91-08, SRI International, Computer Science Laboratory, November 1991.

[25] C. L. Talcott. An actor rewrite theory. This volume.

[26] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE'94, Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe, Athens, Greece, July 1994*, volume 817 of *LNCS*, pages 648–660. Springer-Verlag, 1994.

9 Appendix

```
fmod META <M : Mod> is
sorts OpId VarId Term TermList Label Nat .
```

```
subsort VarId < Term .
subsort OpId < Term .
subsort Term < TermList .

op _[_] : OpId TermList -> Term .
op _,_ :  TermList  TermList -> TermList [assoc] .
op error* : -> Term .
*** meta-apply is a built in function, that takes the meta-representation
*** of a term, a rule label and a natural number in peano representation.
***
*** meta-apply(t, l, n) is evaluated as follows:
*** (1) "t" is converted to the term it represents.
*** (2) this term is fully reduced using the equations
*** (3) the resulting term is matched against all rules with label "l"
***     with matches that fail to satisfy the condition of their rule
*** discarded.
*** (4) the first "n" successful matches are discarded
*** (5) if there is an (n+1)th match, its rule is applied using that
***     match; otherwise "error*" is returned
*** (6) the new term is fully reduced using the equations
*** (7) the resulting term is converted to a meta-term which is returned
op meta-apply : Term Label Nat -> Term .
op z : -> Nat .
op s : Nat -> Nat .
endfm

*** Here we just introduce the specification of STRAT <M : Mod> needed to
*** compute reductions in NIM-WIN
fmod STRAT <M : Mod> is
extending META <M> .
sorts SolTree SolTreeList SolTreeExp StrategyName Strategy StrategyExp .
subsort Term < SolTree .
subsort SolTree < SolTreeList .
subsort SolTree < SolTreeExp .
subsort StrategyName < Strategy .

op ? : -> SolTreeExp .
op ^ : -> SolTree .
op _,_ :  SolTree SolTreeList -> SolTreeList .
op mk :  SolTreeList -> SolTree .
op _{<-_} : SolTree SolTree -> SolTree .
op sols : Term Label Nat -> SolTreeList .
op failure : -> StrategyExp .
op rew_=>_with_ : Term SolTreeExp Strategy -> StrategyExp .
op _andthen_ : StrategyExp Strategy -> StrategyExp .
op idle : -> Strategy .
op _;_ : Strategy Strategy -> Strategy .
op _;;_orelse_ : Strategy Strategy Strategy -> Strategy .
op apply : Label -> Strategy .
```

```
op dk-apply : Label ->  Strategy .
op downleft : -> Strategy .
op up : -> Strategy .
op prunesol : -> Strategy .
op pruneresest : -> Strategy .

var N : Nat . vars T T' T'' : Term .  var L : Label .
var SlT SlT' : SolTree . var SlTL : SolTreeList .
var S S' S'' : Strategy .

eq rew T => ? with S = rew T => ^{<- T} with S .
eq rew T => SlT with (S ; S') = (rew T => SlT with S) andthen S' .

eq rew T => SlT with idle andthen S = rew T => SlT with S .
eq failure andthen S = failure .

eq rew T => SlT with (S ;; S'' orelse S') =
   if rew T => SlT with S == failure then rew T => SlT with S'
     else rew T => SlT with S andthen S'' fi .

eq rew T => SlT{<- T'} with apply(L) =
   if meta-apply(T',L,z)== error* then failure
     else rew T => SlT{<- meta-apply(T',L,z)} with idle fi .

eq rew T => SlT{<- T'} with dk-apply(L) =
   rew T => SlT{<- mk(sols(T',L,z))} with idle .

eq rew T => SlT{<- mk(sols(T',L,N))} with downleft =
   if meta-apply(T',L,N) == error* then failure
   else rew T => SlT{<- mk(^,sols(T',L,s(N)))}
              {<- meta-apply(T',L,N)} with idle fi .

eq rew T => SlT{<- mk(^,SlTL)}{<- T'} with prunesol =
   rew T => SlT{<- mk(SlTL)} with idle .

eq rew T => SlT{<- mk(^,SlTL)}{<- T'} with pruneresest =
   rew T => SlT{<- ^}{<- T'} with idle .

eq rew T => SlT{<- ^}{<- T'} with up = rew T => SlT{<- T'} with idle .
endfm
```

Electronic Notes in Theoretical Computer Science 4 (1996)

Fast matching in combinations of regular equational theories

S. Eker

Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA

Abstract

We consider the problem of efficient term matching, modulo combinations of regular equational theories. Our general approach to the problem consists of three phases: compilation, matching and subproblem solving. We describe a technique for dealing with non-linear variables in a pattern and show how this technique is specialized to several specific equational theories. For matching in an order-sorted setting we discuss an important optimization for theories involving the associativity equation. Finally we sketch a new method of combining matching algorithms for regular collapse theories and give examples that involve the identity and idempotence equations.

1 Introduction

Rewriting logic provides a general way of specifying computational and logical systems [8,9]. An important part of this generality is making equational theories into an explicit parameter of the formalism. Since any computable data type can be equationally axiomatized [2], this allows specification of the data part of a system in a fully general way. We view a rewrite as acting on data structures so axiomatized.

Implementing rewriting logic for the purposes of executable specification presents interesting challenges, because of the generality of the equational theories that should ideally be supported. The present work has been motivated by the implementation of one such system — the Maude interpreter at SRI [3]. However, we believe that the techniques so developed are quite general and will be useful for other rewriting applications.

[1] Supported by a NATO Fellowship administered through the Royal Society, by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114 and by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

© *1996 Elsevier Science B. V.*

While some part of an equational specification can be organized into a confluent and terminating set of rewrite rules, it is often convenient to have some equational theories handled implicitly by working with congruence classes of terms. For practical implementation purposes, we replace these congruence classes by chosen representatives and the matching step of term rewriting is performed by special matching algorithms, particular to the equational theories in use. For Maude it is important that these special matching algorithms be 'plug-compatible' so that the rewrite engine can be extended in a modular way as new matching algorithms are developed.

The general problem of combining matching algorithms for regular equational theories was solved by Nipkow [10]. While elegant from a theoretical standpoint, Nipkow's method, which is essentially based on variable abstraction, is far too inefficient for practical language implementation. In this paper we present selected algorithms and techniques developed for the Maude interpreter's rewrite engine. Experiments with the current version of the rewrite engine incorporating some of these techniques have shown speed-ups over the OBJ3 interpreter of up to three orders of magnitude when rewriting with associative-commutative function symbols, even when all the patterns linear. When non-linear patterns are used speed-ups are more dramatic. On one associative-commutative rewriting problem the OBJ3 interpreter has failed to terminate after over 110 cpu-hours whereas the rewrite engine has succeeded in under 0.4 cpu-seconds running on the same hardware.

Since the matching algorithms we describe are imperative in nature and involve manipulating complex pointer based data structures, trying to force the algorithms into a functional form or presenting imperative pseudo-code would tend to obscure rather than illuminate the techniques involved. Instead we will focus on (abstract versions) of the data structures used and explain informally how they are constructed and manipulated with the aid of diagrams and carefully chosen examples.

1.1 Preliminaries

We work with the set of terms $T_\Sigma(\mathbf{X})$ over a signature Σ and a set $\mathbf{X}$ of variables. We denote the set of ground terms by T_Σ. We measure the size of a term t by counting the number $|t|$ of symbols in t. We denote the set of variables occurring in a term t by $Var(t)$.

An equation is said to be *regular* if every variable that appears on one side also appears on the other. An equation is *collapse-free* if neither side consists soley of a bare variable; otherwise it is a *collapse equation.* The most important regular collapse-free equations are those for associativity and commutativity for a given binary function symbol. The most important regular collapse equations are those for identity and idempotence for a given binary function symbol. A theory is *regular* if all its equations are regular. A theory is *collapse-free* if all its equations are collapse free; otherwise it is a *collapse theory.*

For the purposes of combining matching algorithms we assume that the set of theories are disjoint; i.e., that no function symbol occurs in more than

one theory. Thus, each function symbol can be said to belong to a particular theory. When we have a term $f(\alpha_1, \ldots, \alpha_n)$ where the top symbol of a subterm α_i belongs to a theory other than that of f we call such α_i an *alien* subterm.

Let E be the union of all the theories containing symbols from Σ. Then we write $t =_E t'$ if $E \vdash t = t'$ by the deduction rules of equational logic. We denote the congruence class $\{t' \mid t' =_E t\}$ of t by $[t]_E$. We will write a matching problem in the combination E of theories as $p \leq^?_E s$, for $p \in T_\Sigma(\mathbf{X})$, $s \in T_\Sigma$ where p is called the *pattern* and s is called the *subject.* The goal of a matching problem is to find all distinct (gound) substitutions σ such that $p\sigma =_E s$. Two substitutions σ and σ' are considered distinct if for at least one variable X occurring in p, $\sigma(X) \neq_E \sigma'(X)$.

Our matching algorithms will build matching substitutions one piece at a time, and it will be useful to have the notion of a partial substitution. A partial substitution is a either a set $p \subset \mathbf{X} \times T_\Sigma$ satisfying the property

$$(X, t_1), (X, t_2) \in p \Rightarrow t_1 = t_2 \tag{1}$$

or the special constant *fail*. Let $\mathbf{P}$ be the set of all such partial substitutions. We define union on partial substitutions, $\sqcup : \mathbf{P} \times \mathbf{P} \rightarrow \mathbf{P}$ by

$$p \sqcup q = \begin{cases} p \cup q \text{ if } p \neq \mathit{fail} \text{ and } q \neq \mathit{fail} \text{ and } p \cup q \text{ satisfies (1)} \\ \mathit{fail} \quad \text{otherwise} \end{cases}$$

2 Regular collapse-free theories

If a subpattern p' has a top symbol belonging to some regular collapse-free theory E_1, then for any substitution σ, any $t \in [p'\sigma]_E$ will also have a top symbol belonging to E_1. This property means that if p' is an alien subpattern, it can only match some piece of the subject whose top symbol also belongs to E_1. Thus, we can recursively decompose a matching problem in a combination of regular collapse-free theories into smaller matching problems in individual theories for which we have matching algorithms.

We work with terms that have been normalized with respect to the union E of theories. Normalization can be seen as a function

$$\mathcal{NF} : T_\Sigma(\mathbf{X}) \rightarrow T_\Sigma(\mathbf{X})$$

that for each term t chooses a unique representative from its congruence class $[t]_E$. We assume that a total ordering $>$ on the set of symbols in Σ is given. For each equational theory, $\mathcal{NF}$ together with a total ordering on normal forms headed by the same function symbol (from that theory) must be defined. For terms that are just variable symbols or free constant symbols, $\mathcal{NF}$ is the identity function and the total ordering is the given ordering $>$ on symbols. For terms in normal form with differing top symbols, the total ordering on normal forms is given by the ordering on their top symbols.

Our basic approach to matching consists of three phases: compilation, matching and subproblem solving. We make the assumption that the same pattern p will be matched against many times with different subjects, and therefore it is worthwhile compiling it.

2.1 Compilation phase

In the compilation phase the normalized pattern is analyzed and a matching automaton is generated. We view compilation as a mapping

$$compile : T_\Sigma(\mathbf{X}) \times \mathcal{P}(\mathbf{X}) \rightarrow \mathbf{A}$$

where $\mathbf{A}$ is the set of *matching automata.* Matching automata are built up in a hierarchical way with larger automata containing subautomata. The actual form of a matching automaton depends on the equational theory in which the matching will be done; thus for an equational theory E_1 we will have E_1-automata. The hierarchical structure of a matching automaton closely reflects the structure of the pattern it was compiled from: where the p has a top symbol in theory E_1 and a pair of alien subterms in theories E_2 and E_3 for example, the compilation of p will be an E_1-automaton containing an E_2-automaton and an E_3-automaton.

The extra argument that *compile* takes is a set of variables whose bindings are guaranteed to be already known at match-time. For the top level compilation of a pattern this will usually be the empty set, however for the compilation of of subterms this extra argument allows some important optimizations as we shall see later. In all the theories we consider in this paper, the size matching of the automata for p will be linear in $|p|$.

2.2 Matching phase

In the matching phase the matching automaton produced from the compilation phase is applied to the normalized subject s. We view matching as a mapping

$$match : \mathbf{A} \times T_\Sigma \rightarrow \mathbf{P} \times \mathbf{S}$$

where $\mathbf{S}$ is the set of *subproblem objects.* Here the result of a match is a partial substitution which contains variables that can easily be determined to have the same value in all matching substitutions will together with a subproblem object which is a compact representation of the possible values for the variables not mentioned in the partial substitution. For a simple pattern the partial substitution might contain bindings for all the variables in the pattern in which case the empty subproblem object denoted by $\oslash$ is returned. Of course the matching phase could fail altogether in which case the pair $(fail, \oslash)$ is returned.

Subproblem objects are built up in a hierarchical way with larger subproblem objects containing subproblem sub-objects. As with matching automata the actual form of a subproblem object depends on the equational theory in which the matching was done; thus an E_1-automaton that could not uniquely bind all the variables occuring in the subpattern it was compiled from will construct an E_1-subproblem object to encode the alternatives. If the E_1-automaton contained an E_2-automaton then the E_1-subproblem object may contain one or more E_2-subproblem objects. In all the theories we consider in this paper, the matching phase requires at most $O(|p|.|s|)$ time and subproblem object has size at most $O(|p|.|s|)$.

2.3 *Subproblem solving phase*

For many simple patterns this phase will be unnecessary as the matching phase will have uniquely bound all the variables. For more complex patterns we are left with a partial substitution and a subproblem object which may contain nested subproblem sub-objects. In the subproblem solving phase the subproblem object is searched for consistent sets of solutions to the unbound variables; each such set corresponds to a different solution to the original matching problem. We view the searching process as a map

$$\mathit{solve} : \mathbf{P} \times \mathbf{S} \rightarrow \mathbf{P} \times \mathbf{S}$$

which takes a partial solution and a subproblem object and returns either a partial solution (which binds all the variables in the original pattern) and a new subproblem object which encodes the remaining possibilities, or the pair $(\mathit{fail}, \oslash)$ to indicate that there are no further matching substitutions.

Note that it is important that *solve* takes a partial substitution as its first argument. While at the top most level *solve* will always be called with the same partial substitution that was generated by *match*, at lower levels in the hierarchy of subproblem objects, *solve* will be called with different partial substitutions as the search progresses.

For implementation purposes subproblem objects actually contain state information to record which possibilities have already been tried and the returned subproblem object is really the original subproblem object with its state updated. Thus, solutions can be extracted from the subproblem object as needed.

In most of the theories we consider in this paper, matching with non-linear patterns is known to be NP-hard [1] and therefore the time required to find the next solution (or to discover that there is no next solution) may be exponential in the number of variables not bound during the matching phase.

2.4 *Constraint propagation analysis*

The order in which subterms are considered is very important for non-linear patterns. Consider for example the pattern

$$p = f(g(V,W), f(g(W,X), g(Y,Z), V), g(X,Y))$$

where f is a 3-ary free function symbol and g is a commutative function symbol. At first sight it might appear that, given a subject s, finding a matching substitution would require searching for solutions to four commutative matching subproblems that were consistent on the bindings to the non-linear variables V, W, X, Y. In fact, by propagating constraints on variables between the matching subproblems and by considering the subproblems in an optimal order, no searching is necessary. The variable V can be bound uniquely once the subject s is known, as its second occurrence has only free function symbols above it. Once the value of V is known the commutative matching subproblems can be solved directly in the order $g(V,W)$ (binding W uniquely), $g(W,X)$ (binding X uniquely), $g(X,Y)$ (binding Y uniquely) and $g(Y,Z)$ (binding Z uniquely).

This is an example of a general technique applicable to non-linear matching problems which can be used to propagate constraints between variable instances lying in different theories. During the compilation phase we find an 'optimal' order in which to match alien subterms by a process we call *constraint propagation analysis*. Constraint propagation analysis can take time that is exponential in $|p|$.

Constraint propagation analysis makes use of two functions that are closely related to *compile* and which are defined for each equational theory. These are

$$\mathcal{CPA} : T_\Sigma(\mathbf{X}) \times \mathcal{P}(\mathbf{X}) \rightarrow \mathcal{P}(\mathbf{X})$$

and

$$\mathcal{EMF} : T_\Sigma(\mathbf{X}) \times T_\Sigma(\mathbf{X}) \rightarrow \mathbf{B}.$$

Intuitively $\mathcal{CPA}(p, V)$ ("Constraint Propagation Analysis") is the union of V and the set of variables that the matching automaton $compile(p, V)$ would bind uniquely at match-time regardless of the subject. For instance in the above example $\mathcal{CPA}(g(V, W), \{V\}) = \{V, W\}$. For all the matching algorithms we consider in this paper $\mathcal{CPA}$ has the following very useful monotonicity property:

$$\mathcal{CPA}(t, V_1) \cup \mathcal{CPA}(t, V_2) \subseteq \mathcal{CPA}(t, V_1 \cup V_2).$$

Intuitively $\mathcal{EMF}(p, t)$ ("Early Match Fail") returns true if for any set of variables V and any substitution σ, $match(compile(p, V), t\sigma) = (fail, \emptyset)$; in other words any automaton for p will always return failure when attempting to match any instance of t. Note that this is stronger than saying that p and t are not unifiable (assuming that $Var(p) \cap Var(t) = \emptyset$). In practice this function can be defined in a conservative way; it is always okay for $\mathcal{EMF}$ to return false and as worst we may miss a potential optimization in the construction of an automaton. An easy way to defined $\mathcal{EMF}$ would be $\mathcal{EMF}(p, t)$ is true if and only if p and t have top symbols in different collapse-free theories. Because of this we will not consider the definition of $\mathcal{EMF}$ any further. We will say that two patterns p_1 and p_2 are *match-independent* if and only if $\mathcal{EMF}(p_1, p_2)$ and $\mathcal{EMF}(p_2, p_1)$ are both true (the definition of $\mathcal{EMF}$ need not be commutative).

3 Particular theories

We now informally describe the normal forms, orderings, automata, constraint propagation functions and subproblem objects used for some particular theories.

3.1 Free theory

For a term headed by a free function symbol $f_\emptyset$ we define the normal form by

$$\mathcal{NF}(f_\emptyset(\alpha_1, \ldots, \alpha_n)) = f_\emptyset(\mathcal{NF}(\alpha_1), \ldots, \mathcal{NF}(\alpha_n)).$$

If $\alpha = f_\emptyset(\alpha_1, \ldots, \alpha_n)$ and $\beta = f_\emptyset(\beta_1, \ldots, \beta_n)$ are in normal form then $\alpha > \beta$ iff there exists $i \in \{1, \ldots, n\}$ such that $\alpha_i > \beta_i$ and for $j \in \{1, \ldots, i-1\}$,

$\alpha_j = \beta_j$. Because the free theory has no equations, all free functions symbols can be considered to belong to the same theory and a single automaton can be used to match a whole free skeleton; i.e. that part of a term obtained by ignoring all subterms headed by non-free symbols. Consider a pattern

$$t = c[X_1, \ldots, X_n, \alpha_1, \ldots, \alpha_m]$$

where c is a free skeleton, $X_1, \ldots, X_n$ are variables and $\alpha_1, \ldots, \alpha_m$ are alien subterms. We now sketch the form of $compile(t, V)$ and $\mathcal{CPA}(t, V)$. Let $\phi : \{1, \ldots, m\} \rightarrow \{1, \ldots, m\}$ be a permutation. We define

$$V_0^\phi = V \cup \{X_1, \ldots, X_n\},$$
$$V_i^\phi = \mathcal{CPA}(\alpha_{\phi(i)}, V_{i-1}^\phi) \text{ for } i \in \{1, \ldots, m\}.$$

Intuitively V_m^ϕ is the set of variables that could be uniquely bound if we built our automaton so as to match the aliens subterms in the order given by the permutation ϕ. We can compare the merits of two permutations ϕ and ψ by comparing $|V_m^\phi|$ and $|V_m^\psi|$. In principle we can find an 'optimal' permutation by searching through all permutations to find one which uniquely binds at least as many variables as any other. In practice the monotonicity property of $\mathcal{CPA}$ allows us to dramatically prune the search space in most cases.

Having selected an optimal permutation ϕ we can put $\mathcal{CPA}(t, V) = V_m^\phi$ and generate a sequence $A_1, \ldots, A_m$ of subautomata for $\alpha_{\phi(1)}, \ldots, \alpha_{\phi(m)}$ by

$$A_i = compile(\alpha_{\phi(i)}, V_{i-1}^\phi) \text{ for } i \in \{1, \ldots, m\}.$$

Then the automaton $compile(t, v)$ will consist of: a sequence of positions (relative to the top of the free skeleton) and free symbols to check; a sequence of positions and variables $(X_1, \ldots, X_n)$ to bind (or check for a clash if already bound); and a sequence of positions and alien automata $(A_1, \ldots, A_n)$ to apply to the subject.

Since each of the alien automata could return a subproblem object, a free theory subproblem object consists of a sequence of alien subproblem objects to be solved in order with backtracking on failure.

3.2 Commutative theories

For a term headed by a commutative function symbol f_C we define the normal form by

$$\mathcal{NF}(f_C(\alpha_1, \alpha_2)) = \begin{cases} f_C(\mathcal{NF}(\alpha_1), \mathcal{NF}(\alpha_2)) \text{ if } \mathcal{NF}(\alpha_1) > \mathcal{NF}(\alpha_2) \\ f_C(\mathcal{NF}(\alpha_2), \mathcal{NF}(\alpha_1)) \text{ otherwise} \end{cases}$$

If $\alpha = f_C(\alpha_1, \alpha_2)$ and $\beta = f_C(\beta_1, \beta_2)$ are in normal form then $\alpha > \beta$ iff either $\alpha_1 > \beta_1$ or $\alpha_1 = \beta_1$ and $\alpha_2 > \beta_2$.

Consider a pattern $t = f_C(\alpha_1, \alpha_2)$ We now sketch the form of $compile(t, V)$ and $\mathcal{CPA}(t, V)$. There are several cases to consider:

Suppose that $Var(\alpha_1) \subseteq V$. Then at match-time, since all the variables in V will already be bound we know exactly what α_1 will match against. Thus the uncertainly introduced by the commutativity equation is removed and we can put $\mathcal{CPA}(t, V) = \mathcal{CPA}(\alpha_2, V)$. A symmetric case occurs if $Var(\alpha_2) \subseteq V$.

Suppose that α_1 and α_2 are match-independent aliens. Then α_1 and α_2 will only be able to match to corresponding aliens in the subject in at most one way and we can discover which way at match-time since when we try the wrong way we are guaranteed failure without having to wait until the subproblem solving phase. Thus the uncertainty introduced by the commutativity equation is removed and we have a choice of two values for $\mathcal{CPA}(t, V)$:

$$\mathcal{CPA}(\alpha_2, \mathcal{CPA}(\alpha_1, V)) \text{ or } \mathcal{CPA}(\alpha_1, \mathcal{CPA}(\alpha_2, V))$$

depending on which of the aliens we match first; we chose the order that gives us the largest set of uniquely bound variables.

If the above cases do not apply then we cannot guarantee to uniquely bind any additional variables during the match phase and we have $\mathcal{CPA}(t, V) = V$.

A commutative theory automaton consists of the top symbol to check (f_C), a sequence of at most two variables and a sequence of at most two alien automata. A commutative theory subproblem object consists of at most two sequences of alien subproblem objects (one for each possible way around to match the alien patterns against alien subjects), each of which is of length at most two and is accompanied by a local partial substitution.

3.3 Associative theories

In order to construct a convenient normal form for terms in associative theories we allow associative function symbols to be variadic. For a term headed by an associative function symbol f_A we define the normal form by

$$\mathcal{NF}(f_A(\alpha_1, \alpha_2)) = f_A(\alpha_1', \alpha_2')$$

where

$$\alpha_i' = \begin{cases} \gamma_1, \ldots, \gamma_n \text{ if } \mathcal{NF}(\alpha_i) = f_A(\gamma_1, \ldots, \gamma_n) \text{ for some } \gamma_1, \ldots, \gamma_n \\ \mathcal{NF}(\alpha_i) \quad \text{otherwise} \end{cases}$$

If $\alpha = f_A(\alpha_1, \ldots, \alpha_n)$ and $\beta = f_A(\beta_1, \ldots, \beta_m)$ are in normal form then $\alpha > \beta$ iff there exists $i \in \{1, \ldots, n\}$ such that for $j \in \{1, \ldots, \min(i-1, m)\}$, $\alpha_j = \beta_j$ and either $i > m$ or $\alpha_i > \beta_i$.

Consider a term $t = f_A(\alpha_1, \ldots, \alpha_n)$. We now sketch the form of $compile(t, V)$ and $\mathcal{CPA}(t, V)$. The key idea is to partition the sequence $\alpha_1, \ldots, \alpha_n$ of subterms into three subsequences, r_1, q, r_2 where r_1 and r_2 will be collectively called the *rigid part* and q will be called the *flex part.* The intuition is that at match time, given a subject $f_A(s_1, \ldots, s_m)$, for each α_i in the rigid part we will be able to determine uniquely what part of the subject it matches.

The partitioning of $\alpha_1, \ldots, \alpha_n$ into rigid and flex part is complicated and we will not attempt to define it here. Instead we give a concrete example that illustrates some of the subtleties involved. Consider

$$t = f_A(g(X, Y), Y, h(A), Z, h(B), W, h(X))$$

where g is a commutative function symbol and h is a free function symbol. Then the rigid part will consist of $r_1 = g(X, Y), Y, h(A)$ and $r_2 = h(X)$ and the flex part will consist of $q = Z, h(B), W$. The reasoning behind this

partitioning is as follows: Consider matching a subject $f_A(s_1, \ldots, s_m)$. We know that $g(X, Y)$ must match s_1 and $h(X)$ must match s_m. Furthermore if we match $h(X)$ we will uniquely bind X (since $\mathcal{CPA}(h(X), \emptyset) = \{X\}$) and then matching $g(X, Y)$ we will uniquely bind Y (since $\mathcal{CPA}(g(X, Y), \{X\}) = \{X, Y\}$). Once we have a binding for Y we know which s_i must match $h(A)$ and we can uniquely bind A. At this point we cannot make any further guarantees about what variables will be uniquely bound at match-time since Z and W could match any number of alien subjects and there may be a whole set of solutions. In this example we assumed an empty set of variables with guaranteed unique bindings at the outset so we have

$$\mathcal{CPA}(t, \emptyset) = \{A, X, Y\}.$$

If we knew that W would be uniquely bound at match time (i.e. if t was part of a larger pattern that would uniquely bind W) then the rigid part would consist of the entire sequence $\alpha_1, \ldots, \alpha_n$ and the flex part would be empty. Consequently

$$\mathcal{CPA}(t, \{W\}) = \{A, B, W, X, Y, Z\}.$$

An associative automaton consists of the top symbol to check (f_A), the compiled rigid part and the compiled flex part. The both the rigid and flex part compile to a sequence of variables and alien automata. The the compiled rigid part contains additional information about what order its members should be matched in. As a further optimization sequences of consecutive alien terms in the flex part may be analyzed using the $\mathcal{EMF}$ function to calculate a shift table similar to one of those used in the Boyer-Moore string matching algorithm.

In general the flex part may match the remaining part of the subject in may ways and generate an associative subproblem object. We now illustrate the structure of an associative subproblem object with the flollowing example. Consider the flex part $q = W, \alpha, X, \beta, Y, \gamma, Z$. Let $\alpha_1, \alpha_2, \alpha_3$ be subterms matching α, $\beta_1, \beta_2, \beta_3, \beta_4$ be subterms matching β, $\gamma_1, \gamma_2, \gamma_3$ be subterms matching γ and $\delta_1, \delta_2, \delta_3$ be other alien subterms. Suppose after matching to rigid part the following sequence of subject terms is left:

$$\delta_1, \alpha_1, \alpha_2, \beta_1, \beta_2, \beta_3, \gamma_1, \alpha_3, \gamma_2, \beta_4, \delta_2, \gamma_3, \delta_3$$

Notice that if α is matched against α_1 then it is possible for β to be matched against any of $\beta_1, \beta_2, \beta_3$ where as if α is matched against α_2 then matching β against β_1 is not possible as X must be assigned at least one subject. In general the possible combinations of matches for alien patterns α, β, γ can be compactly represented as a directed acyclic graph (dag) as shown in Fig. 1.

Here the node α_1 for example will hold the local partial substitution and subproblem object generated by matching α against α_1. Each potential solution to the associative matching problem corresponds to a path from *start* to *end* in the graph. Notice that dags representing matching problems in this way have a special form; for a given patten α if the node representing some matching subject α_j has an arc going to some node n then the node representing any earlier matching subject α_i will have an arc going to n. This property

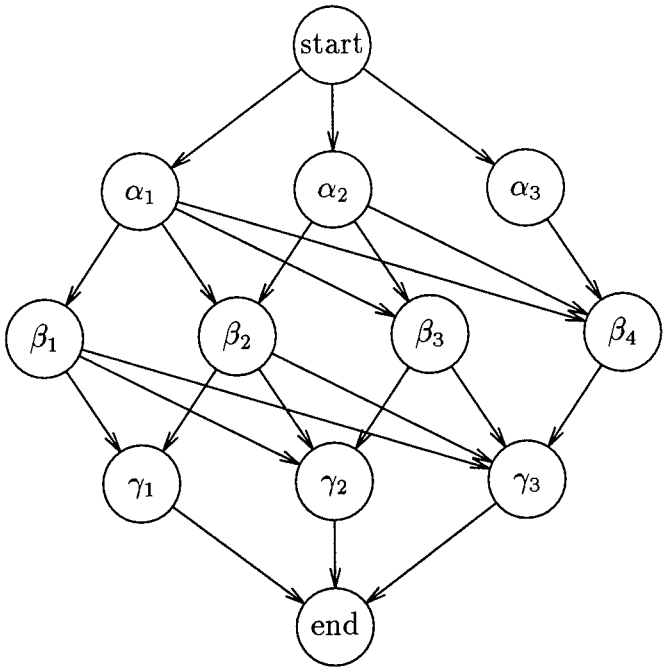


Fig. 1. Dag representing associative subproblem

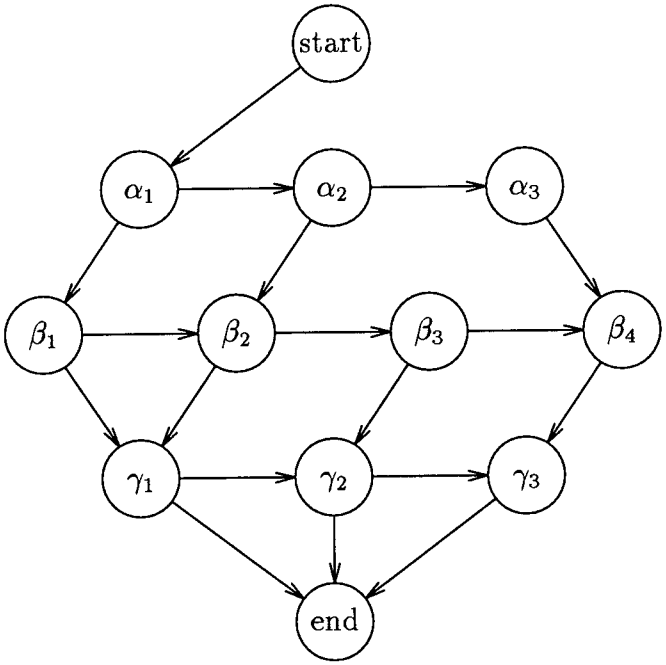


Fig. 2. Simplified dag representing associative subproblem

allows the alternative more compact representation shown in Fig. 2.

Again each path through the dag from *start* to *end* represents a potential solution to the associative matching problem. However here every node has at most two arcs, a horizontal and a vertical one. Taking a vertical arc corresponds to committing to an alien subject (represented by the source node) for some alien pattern where as taking a horizontal arc corresponds to skipping that subject. This structure and *solve* algorithm for it is similar to that

described in [4]. Note that in this example we have made the simplifying assumption that the flex part consists of alternating variables and alien subterms.

3.4 Associative-Commutative theories

We use the *ordered normal form* and term ordering introduced by Hullot [7]. Here, we allow associative-commutative function symbols to be variadic, and allow the arguments of an associative-commutative function symbol to take a positive integer *multiplicity* denoted by a superscript. The normal form is obtained by flattening nested occurences of the same associative-commutative function symbol (as we did for associative function symbols), normalizing and sorting the alien subterms, and replacing k identical subterms by a single instance with multiplicity k. For example consider

$$t = f_{AC}(f_{AC}(\alpha_4, \alpha_1), f_{AC}(\alpha_3, \alpha_2))$$

where each α_i is headed by a symbol other than f_{AC}, $\mathcal{NF}(\alpha_2) = \mathcal{NF}(\alpha_4)$ and $\mathcal{NF}(\alpha_1) > \mathcal{NF}(\alpha_2) > \mathcal{NF}(\alpha_3)$. Then $\mathcal{NF}(t)$ is

$$f_{AC}(\mathcal{NF}(\alpha_1), (\mathcal{NF}(\alpha_2))^2, \mathcal{NF}(\alpha_3))$$

Consider the pattern $t = f_{AC}(\alpha_1^{k_1}, \ldots, \alpha_n^{k_n})$ We now sketch the form of $compile(t, V)$ and $\mathcal{CPA}(t, V)$. Since the associative and commutative equations together allow arbitrary permutation of the subterms under an associative-commutative function symbol the possibilities for constraint propagation analysis are severely limited. There are however some special cases.

Suppose that for $i \in \{2, \ldots, n\}$, $Var(\alpha_i) \subseteq V$. Then at match-time we will know exactly what subject terms $\alpha_2^{k_2}, \ldots, \alpha_n^{k_n}$ will match; and once these have been eliminated, whatever is left must match $\alpha_1^{k_1}$. Thus $\mathcal{CPA}(t, V) = CPA(\alpha_1, V)$. There are of course $n - 1$ other symmetric cases.

Suppose that none of the α_i is a variable. Then each α_i will match exactly one alien subterm in the subject. Now if some α_i is match-independent from all the others then we will be able to find a unique match for it (if it matched more than one alien subterm in the subject, match independence would ensure that the match as a whole would fail). We call such an α_i *match-unique*. Now if we take the set of all match-unique subpatterns we can use the same search technique that we described for the free theory to find an optimal order to match them and do constraint propagation analysis.

An associative-commutative automaton consists of the top symbol to check, a set of variables (with multiplicities), a set of ground alien terms (with multiplicities) and a sequence of alien automata (with multiplicities). At match-time ground alien terms can be rapidly cancelled with alien terms in the subject by using binary search. Similarly with the bindings of any bound variables.

We now illustrate the structure of an associative-commutative subproblem object. Suppose that after eliminating ground terms and bound variables we are left with the variables X^2, Y and the alien subterms α^2, β, γ in the pattern and the aliens subterms $\beta_1^2, \beta_2, \gamma_1^3, \gamma_2$ in the subject. Let β match β_1 and β_2;

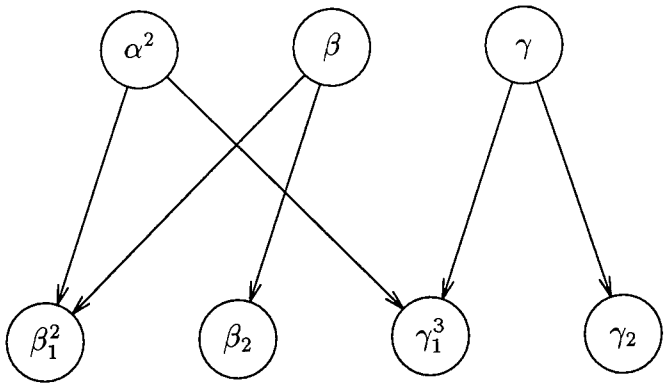


Fig. 3. Bipartite graph representing associative-commutative subproblem

γ match γ_1 and γ_2; and α match all four subject subterms. This situation can be compactly represented by the bipartite graph shown in Fig. 3 and the set of variables (with multiplicities).

Here the arc from the node α^2 to the node β_1^2 will be labelled with the local partial substitution and the subproblem object generated by matching α against β_1 for example. Note that there is no arc from node α^2 to node β_2 because of multiplicity considerations.

An associative-commutative subproblem object consists of such a bipartite graph and a set of variables (with multiplicities). It is solved by a modified form of bipartite graph solving together with Diophantine system solving (to handle the remaining unbound variables). A more detailed description of a similar associative-commutative-matching algorithm (but omitting the compilation phase and including separate Diophantine system solving phase) is given in [5].

4 An optimization for order sorted matching

We now give an important optimization technique for matching modulo associativity (or associativity-commutativity) in an order sorted setting. Consider the following pseudo-Maude code to define a function f on an associative list.

```
fmod LIST is
  sorts Element List .
  subsort Element < List .
  op cons : List List -> List [assoc] .
  op f : List -> List .

var E : Element .
var L : List .
  eq f(E) = ...
  eq f(cons(E, L)) = ...
endfm
```

Here the subpattern $cons(E, L)$ is used to extract a single element from the head of a list. Consider the subject $f(cons(a, b, c, d))$. If sorts are tested

as substitutions are generated, we could be unlucky and generate the sequence of substitutions $\{E \rightarrow cons(a,b,c), L \rightarrow d\}, \{E \rightarrow cons(a,b), L \rightarrow cons(c,d)\}, \{E \rightarrow a, L \rightarrow cons(b,c,d)\}$. The first two substitutions fail because E can only take terms of sort `Element` and any term with *cons* has sort `List` which is larger. Naturally, generating the superfluous substitutions can get extremely expensive for large lists. A similar problem arises with associative-commutative set constructors. While it is possible to solve the problem for simple patterns using ad hoc tricks we present a general method.

Since we are interested in associative (and associative-commutative) function symbols, the domain and range sorts must lie in the same connected sort component and thus, for simplicity, we can assume that the sort structure consists of just a single connected sort component S.

If an order-sorted signature is *pre-regular* [6], for each n-ary function symbol f there exists a order-sorting function

$$\mathcal{S}_f : S^n \rightarrow S$$

which given a tuple of argument sorts $s_1, \ldots, s_n$ yields the least sort of any term $f(\alpha_1, \ldots, \alpha_n)$ where α_i has least sort s_i.

Let f be is an associative (associative-commutative). We insist that $\mathcal{S}_f$ is associative (associative-commutative) otherwise there may not be a well defined least sort for every congruence class. There exists a function

$$\mathcal{B}_f : S \rightarrow \{1,2,3,\ldots\} \cup \{\infty\}$$

which for each sort s gives an upper bound on the number of alien subterms that could be assigned to a variable of sort s occurring under f in a pattern. In the above example $\mathcal{B}_{cons}(\texttt{Element}) = 1$ and $\mathcal{B}_{cons}(\texttt{List}) = \infty$. In order to precisely define $\mathcal{B}_f$ we first extend $\mathcal{S}_f : S^2 \rightarrow S$ to a family of functions $\mathcal{S}_f : S^n \rightarrow S$ for $n \geq 2$ by the following recursive definition.

$$\mathcal{S}_f(s_1, \ldots, s_n) = \mathcal{S}_f(\mathcal{S}_f(s_1, \ldots, s_{n-1}), s_n) \text{ for } n > 2.$$

Then $\mathcal{B}_f(s)$ is the least n such that for all $\vec{s} \in S^n$, $\mathcal{S}_f(\vec{s}) \not\leq s$ if such an n exists or ∞ otherwise. To effectively compute $\mathcal{B}_f$ in polynomial time we represent both $\mathcal{S}_f$ and $\mathcal{B}_f$ as arrays indexed by sorts and use the algorithm given in Fig. 4.

The function $\mathcal{B}_f$ can be used during associative (or associative-commutative) matching to avoid generating useless assignments to variables under f which have sort s for some s with $\mathcal{B}_f(s) \neq \infty$. Also when compiling a pattern $f(t_1, \ldots, t_n)$ for an associative function symbol f and subterm t_i which is a variable of sort s for $\mathcal{B}_f(s) = 1$ may be treated as an alien for the purposes of determining the rigid part and constraint propagation since it will only match a single alien in the subject. In the example above the subpattern $cons(E, L)$ compiles to a particularly efficient associative matching automaton which never generates a subproblem object.

```
for s ∈ S do B_f[s] := ∞ od;
n = 1; limit := 1;
while n ≤ limit do
    T = ∅;
    for (s_1, s_2) ∈ S^2 do
        if B_f[s_1] + B_f[s_2] > n then T := T ∪ {s | s ≤ S_f[s_1, s_2]} fi
    od;
    for s ∈ S do
        if s ∉ T ∧ B_f[s] = ∞ then B_f[s] := n; limit := 2 × n fi
    od;
    n := n + 1
od
```

Fig. 4. Algorithm for computing $\mathcal{B}_f$ from $\mathcal{S}_f$.

5 Regular collapse theories

We start with the observation that regular collapse equations have a very restricted form; one side is some bare variable X and the other side may not contain variables other than X. For simplicity we will only consider the two most important collapse equations; identity and idempotence for a given binary function symbol. We will assume than normal forms chosen for collapse theories are such that terms in normal for are fully collapsed; i.e. no application of a collapse equation in the collapse direction is possible. In many cases designing individual matching algorithms for regular collapse theories are not much more difficult than designing algorithms for regular collapse-free theories. In particular, the algorithms for associative and associative-commutative matching can easily be extended to handle associative-identity and associative-commutative-identity matching respectively by allowing the possibility of variables not being assigned any alien subject subterm, but instead taking the identity element. (Although in some other collapse theories, notably associative-idempotent and associative-idempotent-identity theories even computing unique normal forms can be tricky, in this case because of the lack of commutativity.) The real problem arises with the combination of collapse theories. Consider for example the matching problem

$$f(X, u(Y, f(X, Y))) \leq^?_E f(a^2, 1_u)$$

where f is associative-commutative and u has the identity element 1_u. The straightforward decomposition into smaller matching problems which we used for collapse-free theories fails here, because of what we call an *inter-theory collapse*; when Y is assigned 1_u, the subpattern $u(Y, f(X, Y))$ collapses to $f(X, 1_u)$ whose top symbol belongs to the enclosing theory.

The OBJ3 interpreter gets around this problem by adding additional conditional equations to the rewrite system using a process called *identity completion*. Our approach is to do bottom-up preprocessing on the patterns before compiling them, making explicit the possible inter-theory collapses by the generation and propagation of *OR-nodes*.

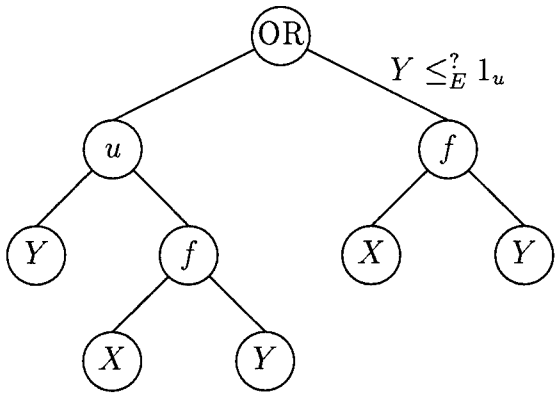


Fig. 5. OR-tree for $u(Y, f(X, Y))$

.5.1 *OR-nodes*

We consider terms as ordered trees, with nodes labeled by elements of $\Sigma \cup \mathbf{X}$ and arcs unlabelled. Preprocessing will introduce OR-nodes into the tree. An OR-node is a special node that has two or more labelled arcs to alternative subtrees — intuitively these represent different possible collapses of a given subpattern while the labels on the arcs represent the conditions necessary for the collapses to take place. Each OR-node arc label consists of a (possibly empty) conjunction of matching problems $p_1 \leq_E^? s_1 \wedge \cdots \wedge p_n \leq_E^? s_n$. The result of matching one of the alternative subtrees of an OR-node is only valid if the conjunction of matching problems on the corresponding arc can be solved (binding the variables that were eliminated by the collapse). OR-nodes are introduced into a the tree representing a pattern when the possibility of a collapse is recognized. An OR-node is propagated up the tree when one of its subtrees interacts with (i.e. has its top fucntion symbol in the same theory as that of) its parent node.

The preprocessed pattern forms an *OR-tree.* When an OR-tree is compiled, the OR-nodes compile to OR-automata which have a subautomaton for the subterm below each arc together with additional subautomata to deal with the conjunctions of matching problems labelling the arcs. At match-time OR-automaton may give rise to OR-subproblem objects. We now illustrate the technique with two examples.

First we consider the pattern $f(X, u(Y, f(X, Y)))$ from the matching problem example above. Working bottom-up $f(X, Y)$ cannot collapse so preprocessing leaves it unchanged. The possibility of a collapse occurs with the subpattern $u(Y, f(X, Y))$. Preprocessing introduces an OR-node with two alternatives as shown in Fig. 5.

In the left alternative no collapse happens and the subpattern remains unchanged. The right alternative is where $Y = 1_u$ and the subpattern collapses to $f(X, Y)$. The condition $Y = 1_u$ appears as a matching problem on the right arc; although here the condition is particularly simple and does not need the full generality of matching. Moving up the pattern we see that there is the possibility of an interaction between the top f function symbol and the f in the right alternative of the OR-node, so the OR-node must be propagated

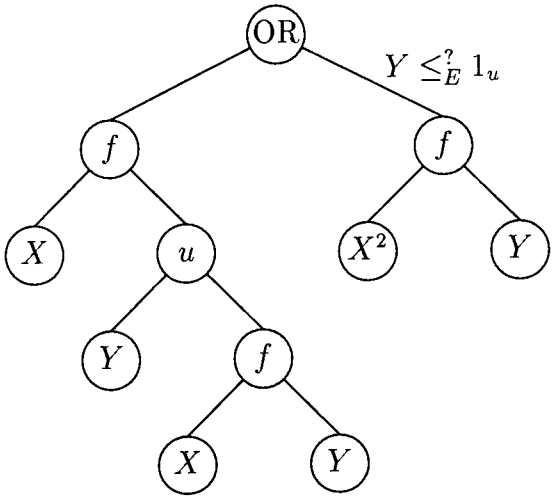


Fig. 6. OR-tree for $f(X, u(Y, f(X,Y)))$

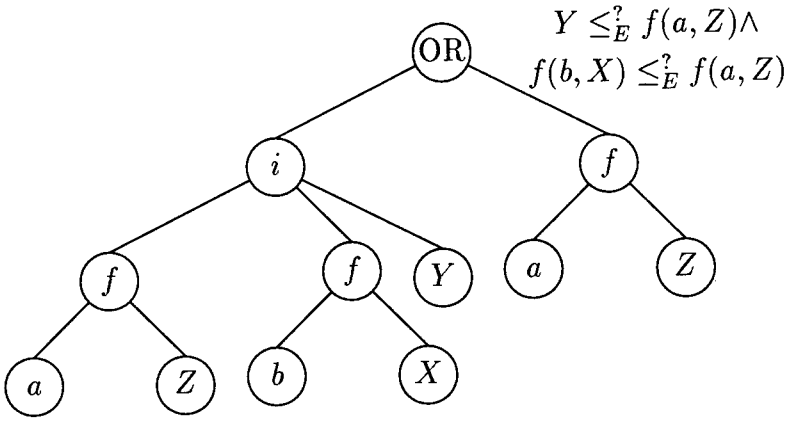


Fig. 7. OR-tree for $i(f(a,Z), f(b,X), Y)$

upwards, yielding the OR-tree shown in Fig. 6.

In the left alternative in Fig. 6 where no collapse takes place and the original pattern appears below the arc. The right alternative is where the collapse takes place and the two f symbols combine by flattening; the condition on the right arc of the OR-node has not changed.

We now give a more complex example which illustrates the need for conjunctions of matching problems labelling the OR-node arcs. Consider the pattern $f(a, i(f(a,Z), f(b,X), Y), X)$ where where f is associative-commutative and i is associative-commutative-idempotent. Working bottom-up the possibility of a collapse occurs with the subpattern $u(Y, f(X,Y))$. Preprocessing introduces an OR-node with two alternatives as shown in Fig. 7.

The left alternative is where no collapse happens and the subpattern remains unchanged. The right alternative is where all three subpatterns under the symbol i match the same thing and the subterm headed by i collapses by the idempotence equation. Since in this case all three subpatterns under i match the same thing we can arbitrarily choose which one the subterm headed by i collapses to; we choose $f(a,Z)$. The condition for the collapse to take

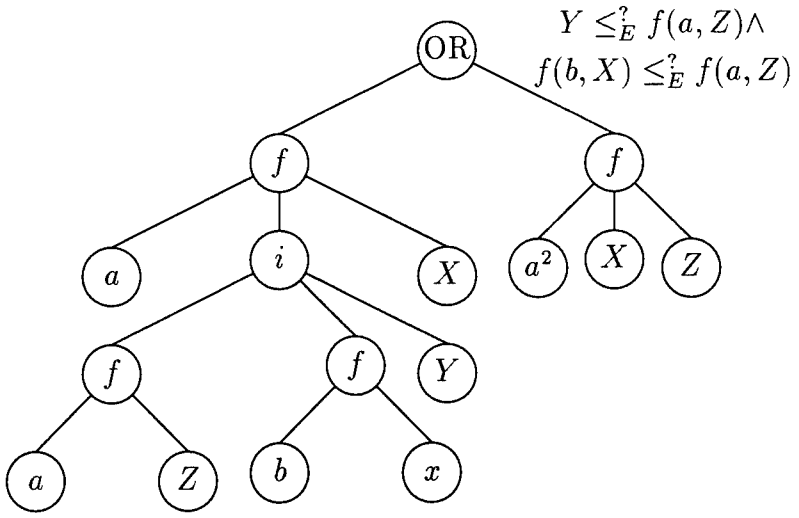


Fig. 8. OR-tree for $f(a, i(f(a,Z), f(b,X), Y), X)$

place is that $Y =_E f(a,Z)$ and $f(b,X) =_E f(a,Z)$. At first sight it may appear that we have introduced a unification problem with this condition. However at match-time when this condition is tested $f(a,Z)$ will already have been matched and Z will be bound to a ground term and hence we can treat $f(a,Z)$ as ground and evaluate the condition using only matching.

In general when a pattern $i(p_1,p_2)$ collapses p_1 due to idempotence during the preprocessing step there are a couple of subtle issues. Firstly since p_1 will ultimately be matched against (part of) the subject (even though it may then be part of a bigger pattern), all the variables in it will ultimately be bound. However they may not be bound uniquely until the subproblem solving phase and thus the condition $p_2 \leq_E^? p_1$ may have to be pushed into an OR-subproblem object. Secondly p_1 and p_2 themselves may already be OR-trees. In this case the collapse branches of p_1 may be discarded in the condition since they correspond to normalization of p_1 after substituting for its variables which can be done by our normal form function.

Moving up the pattern we see that there is the possibility of an interaction between the top f function symbol and the f in the right alternative of the OR-node, so the OR-node must be propagated upwards, yielding the OR-tree shown in Fig. 8.

Here the left alternative of the OR-node is where no collapse takes place and the original pattern appears below the arc. The right alternative is where the collapse takes place and the two f symbols combine by flattening; the condition on the right arc of the OR-node has not changed.

6 Concluding Remarks

We have presented a general approach to matching in combinations of regular equational theories. The key ideas are:

(i) To analyze the pattern before the subject is seen in order to determine the best order to match its parts and to produce an automaton which

includes additional information to accelerate the matching process.

(ii) To have a polynomial time match phase which binds those variables for which it can find unique bindings and constructs a subproblem object to encode the possible bindings of the other variables.

(iii) To confine the potentially exponential searching needed to find consistent sets of solutions for smaller matching problems to a subproblem object whose design is optimized for fast non-recursive exploration of the search space.

(iv) To make use of the sort structure of associative (associative-commutative) function symbols to prune the search space.

(v) To preprocess patterns to make inter-theory collapses explicit.

We notice that the kind of matching problems that arise in algebraic (OBJ-style) programming often differ considerably from those that arise in theorem proving (e.g. using structural induction). The former tend to have simple linear or almost linear patterns for which constraint propagation analysis is useless but huge associative (associative-commutative) subjects (representing data). The latter tend to have complex non-linear patterns but relatively small subjects. Both kinds of matching problems can present difficulties to naive matching algorithms. The algorithms we have discusses are mostly aimed at the latter kind of matching problem. The current version of the rewrite engine lacks collapse theories but includes a second family of algorithms that we call *greedy matching algorithms* that are optimized for generating a single solution to matching problems that have a simple pattern. These provide a small (but useful) constant factor speed up in many cases.

Two problems that we have yet to considered are many-to-one matching and order of generation of matching substitutions. Many-to-one matching is important in term rewriting because one has a single subject and a choice of rewrite rules. The indexing scheme currently used in the rewrite engine is based on [11] and gives the equivalent of many-to-one for the topmost free function symbol skeleton (if any) of each pattern. Going beyond this to combine many-to-one matching with constraint propagation analysis seems particularly difficult.

For many (confluent, terminating) term rewriting systems the choice of matching substitutions used has a marked effect on the number of rewrites needed to bring a term into fully reduced form. Ideally we would like to use global information about the term rewriting system to influence the order in which the matching algorithm generates substitutions. Again this seems difficult.

Acknowledgements

I thank Manuel Clavel, Patrick Lincoln and José Meseguer for many useful discussions on the topics in this paper.

References

[1] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3:203–216, 1987.

[2] J. Bergstra and J. Tucker. A characterization of computable data types by means of a finite equational specification method. In *Automata, Languages and Programming, Seventh Colloquium*, Lecture Notes in Computer Science 81, pages 76–90. Springer-Verlag, 1980.

[3] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. This volume.

[4] S. Eker. Associative matching for linear terms. Technical Report CS-R9224, Center for Mathematics and Computer Science, Amsterdam, July 1992.

[5] S. Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

[6] J. Goguen and J. Meseguer. Order sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[7] J.-M. Hullot. *Compilation de Formes Canoniques dans les Theories Equationnelles.* PhD thesis, University de Paris Sud, Orsay, France, 1980.

[8] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993.

[9] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[10] T. Nipkow. Combining matching algorithms: The regular case. *Journal of Symbolic Computation*, 12:633–653, 1991.

[11] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In *Automata, Languages and Programming, 19th Colloquium*, Lecture Notes in Computer Science 623, pages 247–260. Springer-Verlag, 1992.

Electronic Notes in Theoretical Computer Science 4 (1996)

Distributed Logic Objects: A Fragment of Rewriting Logic and its Implementation

Anna Ciampolini[†], Evelina Lamma[†], Paola Mello[‡], Cesare Stefanelli[† 1]

[†] *DEIS, Università di Bologna*
Viale Risorgimento 2, 40136 Bologna, Italy
[‡] *Istituto di Ingegneria, Università di Ferrara*
Via Saragat, 44100 Ferrara, Italy
{*aciampolini,elamma,pmello,cstefanelli*}*@deis.unibo.it*

Abstract

This paper presents a logic language (called *Distributed Logic Objects*, *DLO* for short) that supports objects, messages and inheritance. The operational semantics of the language is given in terms of rewriting rules acting upon the (possibly distributed) state of the system. In this sense, the logic underlying the language is Rewriting Logic. In the paper we discuss the implementation of this language on distributed memory MIMD architectures, and we describe the advantages achieved in terms of flexibility, scalability and load balancing. In more detail, the implementation is obtained by translating logic objects into a concurrent logic language based on multi-head clauses, taking advantage from its distributed implementation on a massively parallel architecture. In the underlying implementation, objects are clusters of processes, objects' state is represented by logical variables, message-passing communication between objects is performed via multi-head clauses, and inheritance is mapped into clause union. Some interesting features such as transparent object migration and intensional messages are easily achieved thanks to the underlying support. In the paper, we also sketch a (direct) distributed implementation supporting the indexing of clauses for single-named methods.

1 Introduction

Several ways of combining object-oriented and logic programming have been proposed to achieve data abstraction, modularity and code reuse. Some proposals have implemented logic objects on stream-based concurrent logic languages (e.g., [15,23,10]), but this choice is not the best for distribution and scalability. Streams, in fact, behave like shared variables and thus introduce a centralization point in the resulting computational model. In particular, stream communication is programmed by having a producer writing messages

[1] We thank the anonymous referees for their useful comments and the CNR coordinated project on integration of logic and objects.

© *1996 Elsevier Science B. V.*

into a difference list, whose head is read by a consumer. To merge multiple streams, a chain of active *merge* processes is needed, thus requiring extra process reductions and lengthening transmission delay.

In the meanwhile, Meseguer proposed a logic theory of concurrent objects in [19] by defining Rewriting Logic. Rewriting Logic is a very general model of concurrency from which many other models can be obtained by specialization. In this logic, rewriting can take place *modulo* an arbitrary set of structural axioms which could be undecidable. This suggests considering subsets of Rewriting Logic to be efficiently implementable.

For instance, the *Maude* language integrates in a very simple and natural manner functional, object-oriented, relational and concurrent programming by supporting term rewriting, graph rewriting and object-oriented rewriting. In particular, the general form of *Maude* rewrite rules "represents communication events in an object-oriented system where it is possible for one, none, or several objects to appear as participants in the left-hand side of rules" [21].

In a later work [20], Meseguer and Winkler introduce a subset of the *Maude* language called *SimpleMaude*. *SimpleMaude* rules involve only (at most) one object and one method in their left-hand side. This is mainly motivated by the need of having an efficient implementation on a wide variety of parallel architectures, ranging from sequential, SIMD, MIMD, and MIMD/SIMD machines (see [18]).

In this paper, we introduce the language of *Distributed Logic Objects* (*DLO*, for short in the following), that is characterized by active, asynchronously executing agents which communicate through message passing. *DLO* can be considered a particular instance of the general theory of Rewriting Logic where only object-oriented rewriting is supported.

As in [21], the approach we consider for the implementation of *DLO* is translation. The idea is to apply program transformation techniques which are semantics-preserving. In this way, we can allow the full generality of *DLO* even if at the expenses of some efficiency. The target language for transforming *DLO* programs is a concurrent logic language (Rose [7]) with multi-head clauses. In Rose, inter-process communication is performed via multi-head clauses as in [12,22], and *AND*-parallel goals do not share variables in order to avoid centralization points. Rose has been implemented on a parallel architecture based on the transputer technology [8] by extending the abstract machine for Prolog [24] with new instructions and data structures supporting distributed unification, process creation and communication, and control of nondeterminism. In the resulting implementation of *DLO*, we map each logic object into a set of Rose goals and clauses, messages between objects into goal invocations, and object names into logic variables. Furthermore, method definitions are translated into Rose clauses and inheritance is obtained through the notion of clause union.

The translation approach is quite effective, and has been used in the past to implement object-oriented systems on top of concurrent logic languages. By translating distributed logic objects into Rose, we obtain a number of distinguishing features. In particular, since local and remote method invoca-

tions are treated in a uniform way, it is possible to move objects at run-time among the nodes of the distributed system, thus allowing a sort of dynamic load balancing. This makes the real implementation scalable with the underlying architecture. Moreover, object names being mapped into logic variables, intensional messages are easily supported.

The major sources of overhead of the resulting implementation are due to the dynamic creation of remote objects and the broadcasting of messages exchanged through the network. In the transformational approach, broadcasting arises because objects are mapped into logic variables and thus this implementation does employ neither the object addresses nor the inheritance structuring for introducing some kind of "indexing" in selecting methods. We discuss how these sources of overhead can be partially reduced by adopting a direct implementation for a subset of the *DLO* language which corresponds to the fragment of Rewriting Logic where at most one object appears as participant in the left-hand side of clauses.

2 Distributed Logic Objects

The language of *Distributed Logic Objects* aims at integrating the deductive capabilities of logic programming with object-oriented features.

A *DLO* class is a set of (guarded) *DLO* clauses, each one serving some method invocations. *DLO* clauses are multi-head (extended) clauses of the kind:
$\langle M_1, \ldots, M_n \rangle, \langle R_1, \ldots, R_k \rangle, \langle S_1, \ldots, S_m \rangle \leftarrow G | O_1 : m_1, \ldots, O_j : m_j, S'_1, \ldots, S'_q.$
where $q \leq m$, $Pred(S'_1, \ldots, S'_q) \subseteq Pred(S_1, \ldots, S_m)$, $|$ is the commit operator (thus introducing don't care non-determinism), and the guard G is a conjunction of system predicates.

The multi-head of a clause is composed of three multisets of atoms, each one enclosed between angle brackets. The first is the set of atoms (Ms) for methods; the second one (Rs) is for *read-only* state variables, i.e., state variables which do not change their values when the clause is applied; the third one (Ss) is for *mutable* state variables, i.e., variables which possibly change their values because of the clause application.

Atomic goals in the body of a clause $(S'_1, \ldots, S'_q)$ are used for modifying the state of an object. In particular, a rule with a mutable atom in the head and another atom with the same name in the body is a rule for modifying the state of the object. Thus, state changing is obtained through recursive calls to the state of an object. State variables mentioned in the head of a clause as read-only cannot occur in the body, thus preventing their modification.

The introduction of read-only atoms is novel with respect to other proposals grounding logic objects on multi-head clauses [9,5], and avoids passing the state variables of an object to the reinstating recursive call if they are not changed. This feature is not simply syntactic sugar (as in [19], for instance), but it has been specifically introduced at the lower level of the implementation (see section 4) in order to reduce the number of processes created and messages exchanged. It is worth noting, however, that although the *DLO* lan-

guage provides explicit notation for read-only atoms, this optimization could be automatically done at compile time, by statically analysing the code.

In the body of a clause explicit method invocations occur. A goal of the kind $O : M$ corresponds to sending a message M (which is an atom) to the object instance with name O. *self*-method invocations have the form $self : M$. In order to avoid centralization points, no sharing of variables among parallel atomic goals and messages in the body of a *DLO* clause occurs. Only atoms in the body of a *DLO* clause that are executed sequentially can share variables. To this purpose, we have introduced the sequential operator & to make explicit the sequential composition of atoms in the body of a clause. The logical meaning of the parallel conjunction $p(X), q(X)$ (where X is unbound) in the body of a *DLO* clause is the following: $\exists X\ p(X) \wedge \exists Y\ p(Y)$. In other words, the scope of a variable in a parallel conjunction is the single atomic goal as in [7], provided that the variable is not bound to a ground term. This simplify the underlying computational model and, as a consequence, its distributed implementation.

With regard to the communication mode, it can be either synchronous or asynchronous, depending on the kind of goal composition. In fact, in case of a parallel goal (*i.e.*, belonging to a parallel composition) the communication is asynchronous, while in the other case (*i.e.*, sequential goals) the communication is synchronous.

For a *DLO* clause to fire, all its consumable (respectively, read-only) heads have to unify (resp., match) with some messages sent and some state values of the target object. Moreover, the guard evaluation must succeed. When the clause fires, all the messages and the atoms unified with mutable heads are consumed. Then, during the body execution, new goals are possibly created and new messages sent.

Example 2.1 Let us consider the following example, where we adopt the standard Prolog notation for variables:

```
class point::
      <projx>, <>, <y(Y)> ← true | y(0).
      <projy>, <>, <x(X)> ← true | x(0).
      <trans(Dx,Dy)>, <>, <x(X),y(Y)> ←
                        X1 is X+Dx, Y1 is Y+Dy |
                        x(X1), y(Y1).
      <print>, <x(X),y(Y)>, <> ← true |
                        printer:print_values([X,Y]).
```

It represents the code of class `point` of bi-dimensional points. The first clause projects a point on the x-axis. The second clause projects the target point on the y-axis. The third clause applies a rectilinear translation of vector `(Dx,Dy)` to the target point. Notice that to obtain the state change (e.g., setting to zero the y coordinate of the target point), the state variables of the target point (e.g., `y(Y)`) to be modified by the method (e.g., `projx`) must occur both in the head (as mutable atom) and in the body of the clause. The (recursive) occurrence of the state variables in the body thus plays the role of the *become*

primitive of Actor languages [2,3].
The last clause serves a `print` request by raising, in its turn, a `print_values` request to the `printer` object which is a system object [2]. Notice that the coordinates `X` and `Y` of the target point are simply read in the (multi-)head but not consumed, therefore they do not need to be reinstated in the body of the clause.

Thanks to the intrinsic nondeterminism of logic programming languages, different clauses can be written for the same method. At run-time, the adoption of a committed-choice behavior for clause applications will ensure that only one of the definitions is used to serve a method request. For instance, suppose the following clause is added to the class `point` of example 2.1:

```
<print>, <x(X),y(Y)> ← true |
                        laser_printer:print_values([X,Y]).
```

When a `print` message is sent to a target point, only one of the two definitions (and therefore only one of the two printers) will nondeterministically serve the request.

The committed-choice behavior of *DLO* ensures that at most one clause – among those which modify the state variables of an object – will fire. Thus, mutual exclusion is automatically guaranteed in accessing the mutable object state. On the other hand, if the state does not change (the atoms are read-only in the head of clauses) no synchronization is enforced and thus neither is sequentiality.

Distributed Logic Objects have some powerful features usually not present in procedural object-oriented languages, which are inherited from the underlying logic and the logic porgramming paradigm:

- Input and output parameters for methods are not statically fixed but are determined at run-time by using unification. This feature makes *DLO* methods more reusable and flexible.
- *Intensional messages* can be easily supported. A message of the kind `O:print`, where `O` is an unbound variable, is broadcasted to each object of the system. Furthermore, the syntax can be easily extended in order to support multicasting to all the objects of a class. Let us newly consider example 2.1. A message `class(point,O):print`, where `O` is an unbound variable, would be sent to each object of class `point`. Notice that for each message sent (intensional or not) exactly one object will serve the request, due to the committed-choice behavior of method definitions [3]. For instance, the intensional message `class(point,O):print` is sent to every instance of the `point` class but only the first point that commits will produce the

[2] We suppose that there are some globally available objects representing system devices, identified in the program by Prolog constants.

[3] This avoids to keep intensional messages pending for a long time. In fact, an intensional message cannot be discarded because new objects which may consume the message can be created later. However, in *DLO* this message pending must be performend until the commit, thus normally being limited in time.

printing of its state. This feature can be compared to a particular form of *pattern-directed communication* present in Actor systems [3], where an actor can send a message to a single arbitrary member of a group.

- *Multi-named methods* can be defined in the style of Maude [19]. Multi-named methods can be implemented as multi-head clauses with more than one method name in the head. This kind of clauses express a communication event in which different messages from distinct objects participate and synchronize in order to possibly modify the state of a target object and send new messages. For example, the following multi-named method added to the class `point` of example 2.1:

  `<projx, projy>, < >, <x(X), y(Y)>` ← `true | x(0), y(0).`

  synchronizes two messages (`projx` and `projy`) in order to simultaneously set the value of the coordinates of a point to the origin of the `x` and `y` axis.

DLO classes can be connected into hierarchies in order to favour non-replication of behavior. A *DLO* class can inherit part of its instance specification (state variables and behavior) from more general classes (called superclasses). In the following, we will not focus on inheritance (see [1]).

3 Operational Semantics

DLO operational semantics can be given in accordance with the true concurrent model [11] in a way very similar to that presented in [19]. The key idea is to represent the distributed state as a multiset of object states and messages that evolves by concurrent application of rewriting rules. Thus, this semantic description outlines the concurrent distributed nature of the language.

In particular, the state of the system is denoted by a multiset of couples of type $O : A$ (where O is an object name and A is an atom) representing both messages and object state variables. A (renamed apart) multi-head clause, C, of an object O with the form:
$\langle M_1, \ldots, M_n \rangle, \langle R_1, \ldots, R_k \rangle, \langle S_1, \ldots, S_m \rangle \leftarrow G | O_1 : m_1, \ldots, O_j : m_j, S'_1, \ldots, S'_q.$
where $q \leq m$, and $Pred(S'_1, \ldots, S'_q) \subseteq Pred(S_1, \ldots, S_m)$, is interpreted as a rewriting rule. This rule is triggered by a set of messages sent to O and unifying with the method patterns $M_1, \ldots, M_n$ in the head of C. Moreover, read-only state variables $(R_1, \ldots, R_k)$ in the head of C and mutable state variables $(S_1, \ldots, S_m)$ have to be matched and unified with the current values of the variables of the object O, representing (part of) the current distributed state of the system. Finally, the guard G must be successfully evaluated. The outcome of the application of clause C is that messages unifying with $M_1, \ldots, M_n$ and state variables unifying with $S_1, \ldots, S_m$ disappear, the state of the object O changes according to the structure of the new state variables $S'_1, \ldots, S'_q$, and new messages $(O_1 : m_1, \ldots, O_j : m_j)$ are sent (after the application of the unifying substitution).

The following rewriting rule describes the behavior of the object oriented system when a clause is applied. Let $\|O\|$ denote the code of an object, $\lfloor\ \rfloor$ multisets, and $\cup$, $\setminus$ multiset union and difference respectively. Let C_l, $l =$

$1, \ldots, q$ and B_j, $j = 1, \ldots, k$ denote state variables of object O, and A_p, $p = 1, \ldots, n$ be some messages sent to O during the computation. We have:

$$\lfloor O : A_1, \ldots, O : A_n, O : C_1, \ldots, O : C_q, \rfloor \cup \lfloor O : B_1, \ldots, O : B_k \rfloor$$
$$\longrightarrow_r$$
$$\lfloor O_1 : m_1, \ldots, O_k : m_k, O : S'_1, \ldots, O : S'_q \rfloor \theta\gamma \cup \lfloor O : B_1, \ldots, O : B_k \rfloor$$

if the following conditions hold:

- $\exists$ a (renamed apart) clause C belonging to the object O ($C \in \|O\|$);
- $\theta = mgu((A_1, \ldots, A_n, B_1, \ldots, B_k, C_1, \ldots, C_q), (M_1, \ldots, M_n, R_1, \ldots, R_k, S_1, \ldots, S_q))$
- $Eval(G\theta) = \gamma$, where $Eval$ denotes the evaluation of the guard G yielding a computed substitution γ.

Notice that the application of substitutions is component-wise. The substitution $\theta\gamma$ is not applied to the atoms $B_1, \ldots, B_k$ to avoid the creation of bindings for their unbound arguments.

Even if the state of the computation is represented by one single multiset, the rewriting rule applies to a subpart of this multiset which contains elements related to a single object. In this respect, each object constitutes in practice a separate context in a way similar to *Linear Objects* [5].

The computation can be defined in terms of applications of the rewriting rules to disjoint subparts of the current state. Concurrency emerges from the fact that more than one rewriting rule is applied at each step of the computation. The condition to be satisfied in order to simultaneously apply several rewriting rules is that their left- hand sides apply to disjoint sets of mutable elements and messages.

The following rule states how the multiset $\mathcal{S}$ representing the current state of the computation changes because of clause application. Let $\mathcal{S}$ be partitioned into disjoint subparts, X_i, $i = 1, \ldots, h$, and possibly overlapping subparts, R_i, $i = 1, \ldots, h$ (disjoint from X_i). Intuitively, the idea is to permit the parallel application of k clauses (with $k \leq h$) to disjoint subparts of the current state (X_i) which are consumed and rewritten (into subparts Y_i) according to rule $\longrightarrow_r$ and to possibly overlapping subparts of the current state (R_i) which are accessed in read-only mode, and left unchanged by clause application. This behavior is represented more formally by the following rule:

$$\frac{X_i \cup R_i \longrightarrow_r Y_i \cup R_i \ (i = 1, \ldots, k)}{(\mathcal{S} \longrightarrow \mathcal{S} \setminus (X_1 \cup \ldots \cup X_k)) \cup (Y_1 \cup \ldots \cup Y_k)}$$

where $R_i, X_i \subseteq \mathcal{S}$ for $i = 1, \ldots, k$.

Mutual exclusion on the mutable state of an object is automatically guaranteed by the above rule which allows the parallel reduction of clauses only if they do not compete for the same data structure in the current state of the computation.

3.1 Relation with Rewriting Logic

DLO clauses can be interpreted as *rewrite rules* [19]. The outlined *DLO* operational semantics, in fact, corresponds to deduction rules of *Rewriting Logic*. In *Rewriting Logic* deduction is performed by concurrent rewriting modulo structural axioms.

Different types of rewriting are usually considered [18]:

- **term rewriting**, where data structures to be rewritten are *terms*;
- **graph rewriting**, where data structures to be rewritten are *labeled graphs*;
- **object-oriented rewriting**, where data structures to be rewritten are *objects* that interact with each other via asynchronous message-passing.

All these forms of rewriting are supported in the *Maude* language, which integrates in a very simple and natural manner functional, object-oriented, relational and concurrent programming.

In *DLO*, instead, we consider only object-oriented rewriting. As shown in [17], when *Rewriting Logic* is used for object-oriented programming, the structural axioms are associativity, commutativity and identity of a multiset union operator that builds up the configuration of objects and messages. These axioms are implicit in our case, since the order of atoms and messages in a *DLO* clause head is, in practice, not relevant and there exists the identity element *true* with respect to composition of elements in a clause head. Furthermore, as for object-oriented systems based on *Rewriting Logic*, we model the state of the computation as a multiset.

It is worth to notice that, differently from (concurrent) term rewriting [13,16] and object-oriented systems based on *Rewriting Logic* [17], we adopt unification instead of matching for consumable atoms occurring in a clause head. In the case of read-only atoms we use a matching algorithm.

Furthermore, differently from *Rewriting Logic* (and term rewriting systems), in our system congruence in rewriting terms is not present. In fact, in *DLO* it does not happen that rewriting applies to a proper subterm. We use standard unification (or matching, in the case of read-only atoms) algorithm for rewriting terms, thus avoiding the sharing of (nested) data between rewriting clauses which can be a problem in parallel distributed implementations of rewriting systems (see [16,18]).

Like in the language *Maude* [19,21], which is based upon conditional rewriting logic, *DLO* clauses can be conditioned via guards. Thus, *DLO* guarded clauses are equivalent to conditional rewriting rules. However, the commit operator introduced in *DLO* is an extra-logical operator. In fact, through this operator computations are made deterministic. This leads to incompleteness of the resulting logic system, but notably simplifies the implementation avoiding the need for exploring all the alternative subparts of the current state that can be rewritten.

Thanks to the committed-choice nature of *DLO*, each element of the current state of the computation will be rewritten by using at most one clause. Therefore, it is not necessary to follow alternative paths originated by the

application of different clauses to rewrite the same element. Notice that each clause application would possibly assign a different value to the variables of rewritten element.

3.2 Forms of Parallelism

DLO operational semantics outlines the potential parallelism present in the language. The interesting feature is that parallelism has not to be explicitly expressed by the programmer but it is implicitly exploited by the underlying support. As many other concurrent logic programming languages, *DLO* parallelism is fine-grained: this usually implies abundance of potential parallelism. The implicit forms of parallelism exploited in *DLO* can be summarized as follows:

- *inter-object parallelism*: object instances (belonging to the same or to different classes) can execute in parallel since they apply to disjoint sets of atoms. This form of parallelism is inherently related to the *AND*-parallelism of logic programming.
- *intra-object parallelism*: different threads of control can be simultaneously active on the same object. In particular, different methods can be executed in parallel if they do not modify the value of the same state variables, i.e., if they apply to disjoint sets of atoms to be consumed. This is always the case if the object we consider is non-mutable, i.e., all its methods access the object's state variables in read-only mode. In this case, even several applications of the same method for different requests are performed in parallel. If the object we consider is mutable, i.e., some of its methods changes the object's state, the commit operator ensures that only one method at a time changes the state of the object. For example, methods `projx` and `projy` of example 2.1 can be applied in parallel for the same point instance since they do not involve the same variable. The method `trans`, instead, will be executed in mutual exclusion with respect to both `projx` and `projy` since it shares with them part of the mutable state. However, even if two methods cannot be executed in parallel, both multi-head unification and guard evaluation can be performed in parallel. The acceptances of the two invocations of method `trans` and `projx` for an instance of the `point` class are executed in parallel but, after commitment, only one of them will be served.

How to practically support these different forms of parallelism in a distributed system is discussed in section 4.

4 DLO Distributed Implementation

In this section, we describe the main features of the *DLO* implementation on a distributed memory architecture. Distributed memory parallel systems are significantly more problematic than shared memory ones, because of the overhead present when reading and writing nonlocal variables.

The *DLO* programming system is organized into several levels. It allows programs written in the *DLO* language to be compiled and executed on a parallel transputer-based architecture. The distinct parts composing the architectural scheme are:

- The mapping of *DLO* programs into concurrent logic programs. In fact, *DLO* is implemented by following a transformational approach by mapping *DLO* programs into Rose [7] logic programs.
- The run-time environment. The Rose language support consists of a parallel abstract machine which is an extension of the WAM [24]. The parallel abstract machine of the Rose language has been specifically modified to better fit the needs of *DLO* programming, in particluar to support read-only atoms in the head of clauses.
- The physical architecture. It is represented by the MIMD distributed memory architecture, in this case the transputer-based Meiko Computing Surface.

As in [21], the approach we consider for the implementation of *DLO* is translation. The idea is to apply program transformation techniques which are semantics- preserving. In this way, we can allow the full generality of the language even if at the expense of efficiency. The target language for transforming *DLO* programs is a concurrent logic language (Rose [7]) with multi-head clauses. In Rose, inter-process communication is performed via multi-head clauses as in [12,22], and *AND*-parallel goals do not share variables in order to avoid centralization points. Rose has been implemented on a parallel architecture based on the transputer technology [8] by extending the abstract machine for Prolog [24] with new instructions and data structures supporting distributed unification, process creation and communication, and control of non-determinism. In the resulting implementation of *DLO*, we map each logic object into a set of Rose goals and clauses, messages between objects into goal invocations, and object names into logic variables. Furthermore, method definitions are translated into Rose clauses and inheritance is obtained through the notion of clause union.

Example 4.1 Let us consider the class `point` of example 2.1. Its clauses are transformed into the following Rose program P_1:

```
*point(O), projx(O), y(O,Y) ← true | y(O,0).
*point(O), projy(O), x(O,X) ← true | x(O,0).
*point(O), trans(O,Dx,Dy), x(O,X), y(O,Y) ←
                          X1 is X+Dx, Y1 is Y+Dy | x(O,X1), y(O,Y1).
*point(O), print(O), *x(O,X), *y(O,Y) ← true |
                          print_values(printer,[X,Y]).
```

where * is added for denoting read-only atoms.

Notice that objects are represented by Rose predicates (i.e., the "*class*" predicates and the predicates corresponding to the object state), and state change is still achieved by substituting values for the state variables in the

recursive calls to these predicates. However, notice that if a method simply accesses the state of an object for reading values but not for modifying them (e.g., method `print` in class `point`), the predicates corresponding to the object state in the resulting translation occur only in the head of the corresponding Rose clause, being them read-only.

The translation approach is quite effective, and has been used in the past to implement object-oriented systems on top of concurrent logic languages. By translating distributed logic objects into Rose, we obtain a number of distinguishing features. In particular, since local and remote method invocations are treated in a uniform way, it is possible to move objects at run-time among the nodes of the distributed system, thus allowing dynamic load balancing. This makes the real implementation scalable with the underlying architecture. Moreover, object names being mapped into logic variables, intensional messages are easily supported.

Parallelism and Granularity

The transformational approach supports all the forms of parallelism peculiar to *DLO*. The inter-object parallelism is supported by the parallel execution of Rose *AND* processes: object instances can execute in parallel. With regard to intra-object parallelism, two methods corresponds to two Rose clauses which are executed in parallel (at least after the commit phase), provided that they rewrite disjoint subparts of an object state. Therefore, after the commit phase, both the clauses will be able to proceed and execute the method body in parallel.

The adopted forms of parallelism are fine grained, and can be efficiently supported by the tightly coupled parallel architecture considered. The grain of parallelism and the relative need of collecting parallelism depends on the features of the available architecture. On a loosely coupled architecture (e.g., a network of workstations) an efficient implementation might require a kind of *serialization*, in order to combine multiple processes (allocated on the same processor) into one and replace local message sends with predicate calls in a way very similar to what has been done for Actors [2].

Transparency

DLO objects are transparent with regard to parallelism and location. In fact, when developing a *DLO* application the programmer has not to be aware of the real degree of parallelism exploited. Parallelism is implicit, sequentiality can be made explicit by using the sequential conjunctive operator. Mutual exclusion in accessing the state variables of an object is directly guaranteed by the underlying support provided that consumable atoms in the head of *DLO* clauses are used.

Furthermore, it is not necessary to be aware of the physical location of an object in order to send it a message. In particular, invoking a method of an object residing on a remote node has exactly the same effect as if performed locally, except for a performance penalty. Whenever an invocation is made, the underlying implementation transparently determines the location of the method that has to perform the task required.

Replication

Object code is contained in the class, possibly replicated on several nodes. Each method is handled by a Rose manager process. If the method code is replicated on several nodes then more manager processes exist, one for each copy. Each manager process remains idle until some request is sent to it. When a request for a method is sent, the method acceptance phase is executed in parallel by all the manager processes of the invoked method. Finally, the method will be served by the first manager process that successfully executes the commit phase. Obviously, creating copies of classes on several nodes is quite expensive, since each method request must be dispatched to all the copies. In addition, all copies are expected to perform the same computation, thus introducing an increasing of the global computational load. This overhead is however limited to the method *acceptance* phase until the commit. The advantage is that class code replication leads to the replication of the object control thread, although limited to the method acceptance phase. The acceptance phase can successfully terminate if there is a sufficient degree of replication to provide the requested method on at least one working node, thus achieving a *limited* form of fault tolerance. Notice that this feature does not provide a *complete* form of fault tolerance since the object is not entirely replicated. In fact, the state variables of an object accessed by a method in a consumable way are replicated in each node where a manager process for the method has been created but, after the commit, only one copy of these state variables survives (i.e., that allocated on the same node of the process successfully completing the commit phase). Therefore, object state variables move from node to node depending on the selected methods, determining a migration transparent to the user, and controlled by the commit operator.

Communication

When translating *DLO* programs into Rose, we map objects' names into logic variables and this is a technique used in most implementations of logic objects. In this way, the concept of message sending is quite far from message passing in traditional object-oriented languages. A sender does not really send the message to the receiver, but rather includes the identifier of the receiver in the message and posts the message to a blackboard-like structure (the set of current goals) from which the receiver picks it up by using unification. The resulting communication mechanism is flexible, since no explicit communication pattern has to be established. Intensional messages can be directly supported by using, in messages, logical variables in place of constants for objects identifiers, and exploiting broadcasting. This, however, has the drawback of introducing inefficiencies, and motivated the adoption of a different approach based on a direct implementation, which is presented in the next section.

The overhead deriving from broadcast communication and distributed unification can be also reduced – as pointed out in [6,4] – by applying static analysis techniques based on abstract interpretation. In particular, they can be suitable to avoid some unification operations which are subject to failure

and useless communications.

Discussion

Some attempts have been done in order to implement systems based on Rewriting Logic on special purpose machines (see, for instance, [14]). Our purpose, as in [16], is different since it consists in implementing *DLO* in general purpose parallel machines, in particular MIMD distributed memory parallel architectures like networks of Transputers. Other attempts of implementing concurrent rewriting systems (and in particular the language *SimpleMaude*) have been done also for SIMD and MIMD/SIMD architectures [18].

The first prototype developed has allowed to experiment the expressive power of *DLO* (and Rewriting Logic) and its impact on distribution: some nice features of distributed object-oriented systems such as dynamicity, transparency, migration and dynamic load balancing are directly provided and even enhanced in our system, with no need for a special treatment at support level. In this respect, our work can be considered a concrete attempt to implement Rewriting Logic on an MIMD distributed memory architecture.

First experimental results have shown the viability of the approach and its scalability. We have experimented *DLO* for implementing a computational-intensive object-oriented real application in the field of low-level vision [1]. Nonetheless, the translation approach suffers the overheads due to the high cost of dynamic creation of processes and their scheduling, plus the cost of message broadcasting and the cost of distributed unification.

Broadcasting arises because objects are mapped into logic variables and thus this implementation does employ neither the object addresses nor the inheritance structuring for introducing some kind of "indexing" in selecting methods. In the following, we discuss how these sources of overhead can be partially reduced by adopting a direct implementation for a subset of the *DLO* language with single-named methods only.

5 A Direct Implementation

As pointed out in the previous section, there is an efficiency problem with the translation approach, similar to the one present in the distributed implementation of a blackboard-like structure. The run-time support has to perform multicasting (i.e., sending a message to a selected group of machines) or even broadcasting communications (i.e., sending a message to all machines) even if *DLO* messages are point-to-point.

In [20], Meseguer and Winkler introduced a subset of the *Maude* language called *SimpleMaude*. *SimpleMaude* rules involve only (at most) one object and one method in their left-hand side. This was mainly motivated by the need of having an efficient implementation. Having at most one object in the head of a rule allows to treat object identifiers as first class elements, and associate them with specific addresses in the node where the object is located (see also [18]). Moreover, messages can be sent to the object at the corresponding address, thus avoiding broadcasting. Finally, having at most one message in the head

of a rule allows to introduce indexing on inherited clauses.

In this section we briefly discuss a direct implementation for the subset of *DLO* with single-named methods only. The fragment of Rewriting Logic here considered corresponds, in practice, to that underlying *SimpleMaude*.

Object reification

In order to limit broadcasting, we should represent object identifiers as machine oriented effective address-like entities. This can be obtained by *reification*, i.e., the direct mapping of object identifiers into process identifiers of the run-time support. The sending of messages to the object is performed by posting messages at the corresponding address. The broadcasting is substituted by point-to-point message exchanges.

Indexing on inherited clauses

In order to support some kind of "indexing" in selecting methods, we rely on data structures similar to C++ virtual tables. In particular, we associate with each class C a *class virtual predicate table* where the addresses of the methods of C are stored. Each entry in the table is a method name. Associated with a method, there is the address of the clause defining the method. If a method is defined by several clauses, then more than one address is reported. When classes are linked into hierarchies, inheritance can be implemented by building one virtual predicate table for each class. The skeleton of each table is determined during the compilation.

Discussion

The drawback of avoiding the broadcast of messages is a more complex implementation of intensional messages. Nonetheless, as in [20], one process can be created to handle this kind of messages and to broadcast them to each object in the system.

Moreover, the reification of object identifiers adopted by the direct implementation reduces the transparency of *DLO* objects with respect to both parallelism and location. In fact, the state variables of an object O are still mapped into parallel processes which possibly migrate during the computation, but both the server process associated with O and the manager processes of O's methods are allocated on specific nodes and do not migrate during the computation.

6 Conclusions

We have presented an object-oriented language based on Rewriting Logic, and discussed its features with particular reference to its implementation on a distributed parallel architecture. The implementation has been obtained via translation on top of a concurrent logic language with committed-choice multi-head clauses and restricted *AND*-parallelism. First experimental results have shown the viability of the approach and its scalability. Nonetheless, the translation approach suffers of the overhead due to the high cost of dynamic

process creation, message passing among objects, plus the cost of scheduling them, and the cost of distributed unification. A direct implementation has been also proposed for a subset of the language with single-named methods only.

The first prototype developed has allowed to experiment the expressive power of *DLO* and its impact on distribution: some nice features of distributed object-oriented systems such as dynamicity, transparency, migration and dynamic load balancing are directly provided and even enhanced in our system, with no need for a special treatment at support level. In this respect, our work can be considered a concrete attempt to implement (a subpart of) Rewriting Logic on an MIMD distributed parallel memory architecture.

References

[1] A.Ciampolini, E.Lamma, P.Mello, and C. Stefanelli. Distributed Logic Objects. Technical Report DEIS, DEIS - University of Bologna, 1994.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[3] G. Agha, S. Frolund, W.Y. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel & Distributed Technology*, 1(2):3–20, 1993.

[4] J.M. Andreoli, T. Castagnetti, and R. Pareschi. Abstract Interpretation of Linear Logic Programming. In D. Miller, editor, *Proceedings of IEEE Symposium on Logic Programming ILPS93.* The MIT Press, 1993.

[5] J.M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. In D.H.D. Warren and P. Szeredi, editors, *Proc. Seventh International Conference on Logic Programming*, pages 495–510. The MIT Press, 1990.

[6] M. Bourgois, J.M. Andreoli, and R. Pareschi. Extending Objects with Rules, Composition and Concurrency: The LO Experience. Technical Report TR-92-26, ECRC, 1992.

[7] A. Brogi. AND-parallelism without Shared Variables. In D.H.D. Warren and Peter Szeredi, editors, *Proc. Seventh International Conference on Logic Programming*, pages 306–324. The MIT Press, 1990.

[8] A. Brogi, A.Ciampolini, E.Lamma, and P.Mello. The Implementation of a Distributed Model for Logic Programming based on Multiple-headed Clauses. *Information Processing Letters*, 42:331–338, 1992.

[9] J.S. Conery. Logical objects. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth International Conference on Logic Programming*, pages 420–434. The MIT Press, 1988.

[10] A. Davison. Polka: A Parlog Object-oriented Language. Technical Report Internal report, Dept. of Computing, Imperial College, 1988.

[11] P. Degano and U. Montanari. Concurrent Histories: A Basis for Observing Distributed Systems. *J.CSS*, 34:442–461, 1987.

[12] M. Falaschi, G. Levi, and C. Palamidessi. A Synchronization Logic: Axiomatic and Formal Semantics of Generalized Horn Clauses. *Information and Control*, 60:36–69, 1984.

[13] J. A. Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In R. Keller and J. Fasel, editors, *Proceedings of Graph Reduction Workshop*, volume 279 of *LNCS*, pages 53–93, Santa Fe (NM, USA), 1987. Springer-Verlag.

[14] J. A. Gougen. The rewrite rule machine project. In *Proceedings of the 2nd International Conference on Supercomputing*, 1987.

[15] K. Kahn, E.D. Tribble, M.S. Miller, and D. G.Bobrow. Objects in concurrent logic programming languages. In *Proceedings of OOPSLA-86*. ACM Press, Portland (Oregon), 1986.

[16] Claude Kirchner and Patrick Viry. Implementing parallel rewriting. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, volume 590 of *LNCS*, pages 123–138. Springer-Verlag, 1992.

[17] Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G.E. Blelloch, K.M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, pages 309–339. DIMACS Series, Vol. 18, American Mathematical Society, 1994.

[18] Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Livio Ricciulli. Compiling rewriting onto SIMD and MIMD/SIMD machines. In *Proceedings of PARLE'94, 6th International Conference on Parallel Architectures and Languages Europe*, pages 37–48. Springer LNCS 817, 1994.

[19] J. Meseguer. A Logical Theory of Concurrent Objects. In *Proceedings of OOPSLA-ECOOP-90*, pages 101–115. ACM Press, 1990s.

[20] J. Meseguer and T. Winkler. Parallel Programming in Maude. Technical Report CSL-91-08, SRI, 1991.

[21] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[22] L. Monteiro. Distributed Logic: A Theory of Distributed Programming in Logic. Technical report, Universidade Nova de Lisboa, 1986.

[23] E. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing*, 1:25–48, 1983.

[24] D.H.D. Warren. An abstract Prolog instruction set. Technical Report TR 309, SRI International, 1983.

Electronic Notes in Theoretical Computer Science 4 (1996)

Reflection and Strategies in Rewriting Logic

Manuel Clavel and José Meseguer

Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA
{*clavel,meseguer*}*@csl.sri.com*

Abstract

After giving general metalogical axioms characterizing reflection in general logics in terms of the notion of a *universal theory*, this paper specifies a finitely presented universal theory for rewriting logic and gives a detailed proof of the claim made in [6] that rewriting logic is reflective. The paper also gives general axioms for the notion of a strategy language *internal* to a given logic. Exploiting the fact that rewriting logic is reflexive, a general method for defining internal strategy languages for it and proving their correctness is proposed and is illustrated with an example. The Maude language has been used as an experimental vehicle for the exploration of these techniques. They seem quite promising for applications such as metaprogramming and module composition, logical framework representations, development of formal programming and proving environments, supercompilation, and formal verification of strategies.

1 Introduction

Reflection is a very desirable property of computational systems, because a reflective system can access its own metalevel and can in this way be much more powerful, flexible, and adaptable than a nonreflective one. Many researchers have recognized the great importance and usefulness of reflection in programming languages [27,26,29,28,13,10,17], in theorem-proving [30,3,24,11,1,16,8,9], in concurrent and distributed computation [15,22,23], and in many other areas such as compilation, programming environments, operating systems, fault-tolerance, and databases (see [25,12] for recent snapshots of research in reflection).

The goal of this paper is to prove in detail the claim made in [6], namely, that rewriting logic is reflective. We then use this fact to provide semantic

[1] Supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, National Science Foundation Grant CCR-9224005, and by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

© *1996 Elsevier Science B. V.*

foundations for internal strategy languages in rewriting logic, that is, languages that control the computations of rewrite theories but that are themselves definable within rewriting logic.

Since the field of reflection has a wealth of important examples but a dearth of general, formalism-independent, semantic foundations, the notion of "reflective logic" typically is not defined formally but is instead illustrated by example. Under such circumstances, a mathematical proof of our claim is not even a meaningful concept. Therefore, we must first make precise what we mean by a "reflective logic," as an essential prerequisite to the statement and proof of our claim. For this purpose we summarize in Section 2 the metalogical axioms of reflection based on the theory of general logics that we proposed in [6].

The key idea is that a class $\mathcal{C}$ of theories in a logic is reflective if we can find inside the class a *universal theory* U that can simulate all other theories in the class in the sense that there is a function, called a *representation function*,

$$\overline{(_ \vdash _)} : \dot{\bigcup_{T \in \mathcal{C}}} \{T\} \times sen(T) \longrightarrow sen(U)$$

such that for each $T \in \mathcal{C}$, $\varphi \in sen(T)$,

$$T \vdash \varphi \iff U \vdash \overline{T \vdash \varphi}.$$

For rewriting logic the class $\mathcal{C}$ of interest is that of all finitely presentable theories, that is, theories with a finite signature and a finite set of axioms. We construct a finitely presentable rewrite theory U and prove the above equivalence.

The correctness of U can then be used to provide semantic foundations for another topic of great importance in rewriting logic, namely *strategies* that can be used to control the rewriting process. Since the rules in rewrite theory need not be Church-Rosser and may not terminate, the need for controlling the rewrites is much stronger than for functional programs. The great opportunity offered by reflection is to make such strategies *internal* to rewriting logic. This means that strategies can themselves be defined by rewrite rules, and can be reasoned about inside the logic.

Again, to make this concept precise not just for rewriting logic but for a logic in general, in Section 4 we give a somewhat simpler version of the general, formalism-independent, notion of strategy already proposed in [6]. Basically, an *internal strategy language* is a theory-transforming function $\mathcal{S}$ that send each theory T to another theory $\mathcal{S}(T)$ whose deductions simulate controlled deductions of T. In Section 5 we discuss reflection in rewriting logic and give a general method for defining internal strategy languages in a sound and extensible manner. These ideas have been applied to Maude, that is an explicitly reflective rewriting logic language [4].

In the concluding remarks we discuss several promising application areas that reflection and internal strategies open up for rewriting logic languages.

2 Reflection in General Logics

We give here a brief summary of the notion of a universal theory in a logic and of a reflective entailment system introduced in [6]. These notions axiomatize reflective logics within the theory of general logics [20]. We focus here on the simplest case, namely entailment systems. However, reflection at the proof calculus level—where not only sentences, but also proofs are reflected—is also very useful; the adequate definitions for that case are also in [6].

For our present purposes it will be the notions of syntax, of entailment system proposed in [20] that play a crucial role. We present below in summarized form the axioms characterizing these notions. The axioms use the language of category theory, but do not require any acquaintance with categories beyond the basic notions of category, functor, and natural transformation.

2.1 Syntax

Syntax can typically be given by a *signature* Σ providing a grammar on which to build *sentences*. For first order logic, a typical signature consists of a list of function symbols and a list of predicate symbols, each with a prescribed number of arguments, which are used to build up the usual sentences. It is enough to assume that for each logic there is a category **Sign** of possible signatures for it, and a functor *sen* assigning to each signature Σ the set $sen(\Sigma)$ of all its sentences. We call the pair (**Sign**, *sen*) a *syntax*.

2.2 Entailment systems

For a given signature Σ in **Sign**, *entailment* (also called *provability*) of a sentence $\varphi \in sen(\Sigma)$ from a set of axioms $\Gamma \subseteq sen(\Sigma)$ is a relation $\Gamma \vdash \varphi$ which holds if and only if we can prove φ from the axioms Γ using the rules of the logic. We make this relation relative to a signature.

In what follows, $|\mathcal{C}|$ denotes the collection of objects of a category $\mathcal{C}$.

Definition 2.1 [20] An *entailment system* is a triple $\mathcal{E} = (\mathbf{Sign}, sen, \vdash)$ such that

- (**Sign**, *sen*) is a syntax,
- $\vdash$ is a function associating to each $\Sigma \in |\mathbf{Sign}|$ a binary relation $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$ called Σ-*entailment* such that the following properties are satisfied:

(1) *reflexivity*: for any $\varphi \in sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$,
(2) *monotonicity*: if $\Gamma \vdash_\Sigma \varphi$ and $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash_\Sigma \varphi$,
(3) *transitivity*: if $\Gamma \vdash_\Sigma \varphi_i$, for all $i \in I$, and $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, then $\Gamma \vdash_\Sigma \psi$,
(4) $\vdash$-*translation*: if $\Gamma \vdash_\Sigma \varphi$, then for any $H : \Sigma \to \Sigma'$ in **Sign** we have $sen(H)(\Gamma) \vdash_{\Sigma'} sen(H)(\varphi)$. □

itemize

Definition 2.2 [20] Given an entailment system $\mathcal{E}$, its category **Th** of *theories*

has as objects pairs $T = (\Sigma, \Gamma)$ with Σ a signature and $\Gamma \subseteq sen(\Sigma)$. A *theory morphism* $H : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ is a signature morphism $H : \Sigma \to \Sigma'$ such that if $\varphi \in \Gamma$, then $\Gamma' \vdash_{\Sigma'} sen(H)(\varphi)$.

A theory morphism $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$ is called *axiom-preserving* if it satisfies the condition that $sen(H)(\Gamma) \subseteq \Gamma'$. This defines a subcategory $\mathbf{Th}_0$ with the same objects as $\mathbf{Th}$ but with morphisms restricted to be axiom-preserving theory morphisms. □

Note that we can extend the functor *sen* to a functor on theories by taking $sen(\Sigma, \Gamma) = sen(\Sigma)$. Note that we have also a functor $thm : \mathbf{Th}_0 \longrightarrow \mathbf{Set}$ associating to each theory $T = (\Sigma, \Gamma)$ the set $thm(T) = \{\varphi \in sen(\Sigma) \mid \Gamma \vdash_\Sigma \varphi\}$ of its theorems.

2.3 Universal Theories and Reflective Entailment Systems

A reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. Two obvious metatheoretic notions that can be so reflected are theories and the entailment relation $\vdash$. This leads us to the notion of a universal theory. However, universality may not be absolute, but only relative to a class $\mathcal{C}$ of *representable* theories. Typically, for a theory to be representable at the object level, it must have a finitary description in some way—say, being recursively enumerable—so that it can be represented as a piece of language.

Definition 2.3 Given an entailment system $\mathcal{E}$ and a set of theories $\mathcal{C} \subseteq |\mathbf{Th}|$, a theory U is *$\mathcal{C}$-universal* if there is a function, called a *representation function*,

$$\overline{(_ \vdash _)} : \bigcup_{T \in \mathcal{C}} \{T\} \times sen(T) \longrightarrow sen(U)$$

such that for each $T \in \mathcal{C}$, $\varphi \in sen(T)$,

$$T \vdash \varphi \iff U \vdash \overline{T \vdash \varphi}.$$

If, in addition, $U \in \mathcal{C}$, then the entailment system $\mathcal{E}$ is called *$\mathcal{C}$-reflective*. □

Note that in a reflective entailment system, since U itself is representable, representation can be iterated, so that we immediately have a "reflective tower"

$$T \vdash \varphi \iff U \vdash \overline{T \vdash \varphi} \iff U \vdash \overline{U \vdash \overline{T \vdash \varphi}} \ \ldots$$

3 Reflection in rewriting logic

In this section we give the rules of deduction for rewriting logic, define a universal theory for a class of finitely presentable rewrite theories, and prove the correctness of such a theory, establishing that indeed satisfies all the formal requirements that we have stated for a reflective logic in Definition 2.3.

3.1 Rewriting logic

A *signature* in rewriting logic is a pair (Σ, E) formed by a ranked alphabet Σ of function symbols and a set E of Σ-equations. Given a signature (Σ, E), *sentences* of the logic are sequents of the form $[t]_E \longrightarrow [t']_E$, where $[t]$ and $[t']$ are E-equivalence classes of terms. A *theory* in this logic, called a rewrite theory, is a triple[2] $T = (\Sigma, E, R)$ with (Σ, E) a signature, and R a set of sequents called the *rewrite rules* of T.

Given a rewrite theory T, we say that T *entails* a sequent $[t] \longrightarrow [t']$, and write $T \vdash [t] \longrightarrow [t']$, iff $[t] \longrightarrow [t']$ can be obtained by finite application of the following rules of deduction. To indicate that $\{x_1, ..., x_n\}$ is the set of variables occurring in either t or t', we write $t(x_1, ..., x_n) \longrightarrow t'(x_1, ..., x_n)$; also $t(\vec{w}/\vec{x})$ denotes the simultaneous substitution of w_i for x_i in t.

(1) **Reflexivity**. For each $[t] \in T_{\Sigma,E}(X)$,

$$\frac{}{[t] \longrightarrow [t]}$$

(2) **Congruence**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{[t_1] \longrightarrow [t'_1] \quad ... \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, ..., t_n)] \longrightarrow [f(t'_1, ..., t'_n)]}$$

(3) **Replacement**. For each rewrite rule $t(x_1, ..., x_n) \longrightarrow t'(x_1, ..., x_n)$ in T,

$$\frac{[w_1] \longrightarrow [w'_1] \quad ... \quad [w_n] \longrightarrow [w'_n]}{[t(\vec{w}/\vec{x})] \longrightarrow [t'(\vec{w'}/\vec{x})]}$$

(4) **Transitivity**.

$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

Lemma 3.1 *Given a theory* (Σ, E, R), t, $t' \in T_{\Sigma,E}$,

$$(\Sigma, E, R) \vdash [t] \longrightarrow [t'] \iff (\Sigma, \emptyset, R \cup \vec{E} \cup \overleftarrow{E}) \vdash t \longrightarrow t',$$

where, by definition, $\vec{E} = \{t_1 \longrightarrow t_2 \,|\, t_1 = t_2 \in E\}$, $\overleftarrow{E} = \{t_2 \longrightarrow t_1 \,|\, t_1 = t_2 \in E\}$.

3.2 A universal theory for rewriting logic

In this section we introduce a theory U and a representation function $\overline{(_ \vdash _)}$ for encoding pairs consisting of a rewrite theory T in $\mathcal{C}$ and a sentence in it as sentences in U, for $\mathcal{C}$ the class of unconditional and unsorted finitely presentable rewrite theories—that is, theories whose ranked alphabet, and set of rules are all finite.

Without any essential loss of generality we assume that the syntax of those theories is given by operators and variables that are strings of ASCII characters. We also assume that all such theories have standard parenthesized notation. However, to ease readability, in the particular case of the theory U, we will adopt some extra notational conventions.

[2] In the standard treatment of rewriting logic, rules in a rewrite theory T have labels in a set L and are written $l : t \longrightarrow t'$. We omit labels in the present version to simplify the exposition. All that we say below has a straightforward extension to the labelled case.

Moreover, for the sake of simplicity, using Lemma 3.1 we will consider the equations in a theory T in $\mathcal{C}$ as bidirectional sequents.

We first introduce the ranked alphabet Σ_U of operation symbols of U and briefly explain how a representation function for U can be defined. Then, we give the set E_U of equations, and the set R_U of rules of U.

The operation symbols of Σ_U are as follows:

$\Sigma_0 = \text{ASCII} \cup \{ \cdot , \triangleright \}$, where ASCII denotes the set of ASCII characters.
$\Sigma_1 = \{ \text{op}\{_\} , \ \text{var}\{_\} \}$
$\Sigma_2 = \{ __ , \ _[_] , \ _;_ , \ _[\rightarrow _] , \ _[\hookleftarrow _] , \ _[\hookrightarrow _] , \ _ \Rightarrow _ , \ _ \mapsto _ , \ _/_ ,$
$\quad _@_ , \ _\ddot{@}_ , \ \langle _, _ \rangle \}$
$\Sigma_3 = \{ _ :: \langle _, _ \rangle \}$
$\Sigma_5 = \{ _ :: \langle _, _, _, _ \rangle \}$

To ease readability we adopt the following notational convention: the occurrence of n underbar characters in an n-ary operator of Σ_U allows us to display the corresponding expressions with mix-fix syntax, adding parentheses when necessary. Thus, for the operator $_ \mapsto _$ we write $X \mapsto Y$ instead of $_ \mapsto _(X, Y)$. Also, we use in Σ_U some characters that are not strings of ASCII characters.

We next define a representation function $\overline{(_ \vdash _)}$ for encoding pairs consisting of a finitely presentable rewrite theory T and a sentence in it as sentences in U,

$$\overline{(_ \vdash _)} : \bigcup_{T \in \mathcal{C}} \{T\} \times sen(T) \longrightarrow sen(U)$$

The definition of $\overline{(_ \vdash _)}$ is given in a top-down fashion. This will make clear the intended meaning of most of the operation symbols of Σ_U. Note that, to ease readability, we recursively define the representation of theories, rules, terms, etc. using an overloaded function symbol $\overline{(_)}$.

- For $T \in \mathcal{C}$ a finitely presentable rewrite theory, and $t \longrightarrow t'$ a sentence in T, $\overline{(T \vdash t \longrightarrow t')} = (\overline{T} @ * [\rightarrow \overline{t}]) \longrightarrow (\overline{T} @ * [\rightarrow \overline{t'}])$.
- For T a rewrite theory $(\Sigma, R) \in \mathcal{C}$ such that V is the finite set of variables appearing in the rules R, $\overline{T} = \langle \overline{V}, \overline{R} \rangle$.
- For V a set of variables $\{v_1, ..., v_n\}$, $\overline{V} = \overline{v_1}; ...; \overline{v_n}$; for V an empty set, $\overline{V} = \cdot$.
- For R a set of rewrite rules $\{r_1, ..., r_n\}$, $\overline{R} = \overline{r_1}; ...; \overline{r_n}$; for R an empty set, $\overline{R} = \cdot$.
- For r a rewrite rule $(t \longrightarrow t')$, $\overline{r} = (\overline{t} \Rightarrow \overline{t'})$.
- For A a set of assignments $(w_1/x_1, \ldots, w_n/x_n)$, $\overline{A} = \overline{w_1} \mapsto \overline{x_1}; ...; \overline{w_n} \mapsto \overline{x_n}$; for A an empty set, $\overline{A} = \cdot$.
- For t a term $f(t_1, ..., t_n)$, $f \in \Sigma_n$, $n > 0$, $\overline{t} = \text{op}\{\hat{f}\}[\overline{t_1}; ...; \overline{t_n}]$; for t a term c, $c \in \Sigma_0$, $\overline{t} = \text{op}\{\hat{c}\}[\cdot]$; and for t a term v, $v \in Var$, $\overline{t} = \text{var}\{\hat{v}\}[\cdot]$. Note that we have assumed that all operators and variables are strings of ASCII characters.
- For a string of ASCII characters $l = a_1 a_2 ... a_n$, $\hat{l} = a_1.a_2.....a_n$.

We next introduce the set E_U of equations,

$$(x;\cdot) = x = (\cdot;x)$$
$$x;(y;z) = (x;y);z$$

Therefore, the operation symbol '$_;_$' is declared associative with '$\cdot$' as its identity element.

We next introduce by groups the set R_U of rules, and explain their intended meaning. As we shall prove later, U itself belongs to the class $\mathcal{C}$ of finitely presentable rewrite theories and with the representation function $\overline{(_ \vdash _)}$ makes the entailment system of rewriting logic reflective. This means that U reifies the entailment relation, and that the operation symbols, equations, and rules of U can be seen as a specification in rewriting logic of its own rules of deduction, including the congruence and replacement rules. The reflexivity and transitivity rules of deduction for a theory $T \in \mathcal{C}$ are directly mirrored by the reflexivity and transitivity rules of deduction for U.

Note that rewriting logic has also a proof calculus [21]. By extending the definition of U along the lines of [14], so as to make explicit and reify the proofs built up by the deduction process, one can similarly exhibit a finitely presentable universal theory U' making the proof calculus of rewriting logic $\mathcal{C}$-reflective, as defined in [6].

To ease readability, variables in the rules appear in *italics*; this is shorthand notation for the convention that all variables are character strings beginning with a quote and having length at least two, so that no ambiguity may ever arise with the ASCII characters themselves, that are constants of Σ_U. We also introduce the notation $t \longleftrightarrow t'$ to indicate a bidirectional rule, that is, a pair of rules $t \longrightarrow t'$ and $t' \longrightarrow t$.

To reify the rule of congruence we will use contexts and (potential) redexes. In fact, our idea is to use contexts and redexes to combine the rules of congruence and replacement. As a result, a step of reified replacement will be taken in any subterm of a reified subject term. In particular, we use the ASCII character '$*$' and the operation symbol '$_[\rightarrow _]$' in Σ_U. In rule 1 below we use these operation symbols to decompose a term $\overline{t}$ to be rewritten into a context and a potential redex. The intended meaning of rules 1 is to indicate the subterm $\overline{t_1}$ of $\overline{t}$ in which a step of rewriting will be attempted.

$$t@u[\rightarrow p[l;q[l_1];l']] \overset{1}{\longleftrightarrow} t@u[\rightarrow p[l;*;l']][\rightarrow q[l_1]]$$

To reify the rule of replacement we have first to reify the condition for its application. Given a rule $t_1(x_1,...,x_n) \longrightarrow t_2(x_1,...,x_n)$, a replacement can be made in a term t iff $t = t_1(\vec{w}/\vec{x})$. We use the rule 2 below to *set aside* a matching problem between the lefthand side of a rule and a potential redex. For that we have introduced in Σ_U the operation symbol '$_ :: \langle _,_,_,_ \rangle$'. The first argument of '$_ :: \langle _,_,_,_ \rangle$' is a pair formed by a rewrite theory and a term decomposed into a context and a potential redex, to which the matching problem is related; the second and third arguments are the lefthand side of a rule and the potential redex respectively. We will use the operation symbols '$\triangleright$' and '$_[\hookrightarrow _]$' in Σ_U during the matching process of any two terms to simultaneously decompose both, in a similar way that '$*$' and '$_[\rightarrow _]$' are

used in rule 1. But in addition, '$\triangleright$' will divide the lists of subterms of any terms being matched into a list of subterms already matched and a list of subterms that have to be matched. Note that rule 2 sets to empty the lists of subterms already matched of the lefthand side of the rule and the potential redex. The fourth argument consists of a pair built up with the operation symbol '$_/_$'. The first element of this pair represents a set of assignments, and the second a set of variables. As we will see below, this argument is needed to handle the case of nonlinear lefthand sides in which a variable can have several occurrences. Note that rule 2 sets to empty the initial set of assignments, and sets the initial set of variables to the set of variables of the theory. Finally, the fifth argument is the righthand side of the selected rule. As we shall see below, this allows us to continue the reified replacement process without having to keep track of the rule that we have selected.

$$\langle v\ ,\ r; (p_1[l_1] \Rightarrow q); r' \rangle @ u[\rightarrow p[l]] \xrightarrow{2}$$
$$\langle v\ ,\ r; (p_1[l_1] \Rightarrow q); r' \rangle \ddot{@} u[\rightarrow p[l]] :: \langle \triangleright[\hookrightarrow p_1[\triangleright; l_1]]\ ,\ \triangleright[\hookrightarrow p[\triangleright; l]]\ ,\ \cdot/v\ ,\ q \rangle$$

The matching problem between any terms $\overline{t}$ and $\overline{t_1}$ will be handled by rules 3–7 below. As expected, they will try to come out with a set of assignments A, such that $\overline{t}$ substituted by A is equal to $\overline{t_1}$. The reified matching process can be seen as a recursive process trying to identify $\overline{t}$ and $\overline{t_1}$ while keeping track of their differences in A. In particular, the rules 3 and 6 indicate that any terms with the same top operator will match only if all their subterms match, and the matching of these subterms will be attempted from left to right. To handle the case of non-linear lefthand sides, we use the pair A/V. The idea is to keep the set A of variables already assigned and the set V of variables not yet assigned disjoint from each other. Note that the initial set V is the set of variables in the theory. When in the reified matching process we reach the base case of matching a variable $\overline{x_i}$ and a term $\overline{t}$, we consider two cases: rule 4 when $\overline{x_i}$ is in V, and rule 5 when $\overline{x_i}$ is in the set of variables in A, because $\overline{x}$ has already been encountered in another occurrence during the matching. We use the operation symbol '$_[\leftarrow _]$' in Σ_U to indicate that a subterm has been succesfully matched. Then, rule 7 simultaneously inserts the subterms successfully matched at the end of the corresponding list.

$$z :: \langle u[\hookrightarrow \mathrm{op}\{s\}[l; \triangleright; p_1[l_1]; l']]\ ,\ u[\hookrightarrow \mathrm{op}\{s\}[l; \triangleright; p[l]; l'']]\ ,\ a/v\ ,\ q \rangle \xrightarrow{3}$$
$$z :: \langle u[\hookrightarrow \mathrm{op}\{s\}[l; \triangleright; l']][\hookrightarrow p_1[\triangleright; l_1]]\ ,\ u[\hookrightarrow \mathrm{op}\{s\}[l; \triangleright; l'']][\hookrightarrow p[\triangleright; l]]\ ,\ a/v\ ,\ q \rangle$$

$$z :: \langle u[\hookrightarrow \mathrm{var}\{x\}[\triangleright; \cdot]]\ ,\ u[\hookrightarrow p[\triangleright; l]]\ ,\ a/(v; (\mathrm{var}\{x\}[\cdot]); v')\ ,\ q \rangle \xrightarrow{4}$$
$$z :: \langle u[\hookleftarrow p[l]]\ ,\ u[\hookleftarrow p[l]]\ ,\ ((p[l] \mapsto \mathrm{var}\{x\}[\cdot]); a)/(v; v')\ ,\ q \rangle$$

$$z :: \langle u[\hookrightarrow \mathrm{var}\{x\}[\triangleright; \cdot]]\ ,\ u[\hookrightarrow p[\triangleright; l]]\ ,\ (a; (p[l] \mapsto \mathrm{var}\{x\}[\cdot]); a')/v\ ,\ q \rangle \xrightarrow{5}$$
$$z :: \langle u[\hookleftarrow p[l]]\ ,\ u[\hookleftarrow p[l]]\ ,\ (a; (p[l] \mapsto \mathrm{var}\{x\}[\cdot]); a')/v\ ,\ q \rangle$$

$$z :: \langle u[\hookrightarrow \mathrm{op}\{s\}[l; \triangleright]]\ ,\ u[\hookrightarrow \mathrm{op}\{s\}[l; \triangleright]]\ ,\ a/v\ ,\ q \rangle \xrightarrow{6}$$
$$z :: \langle u[\hookleftarrow \mathrm{op}\{s\}[l]]\ ,\ u[\hookleftarrow \mathrm{op}\{s\}[l]]a/v\ ,\ q \rangle$$

$$z :: \langle u[\hookrightarrow \mathrm{op}\{s\}[l;\triangleright;l']][\hookleftarrow p]\ ,\ u[\hookrightarrow \mathrm{op}\{s\}[l;\triangleright;l'']][\hookleftarrow p]\ ,\ a/v\ ,\ q\rangle \xrightarrow{7}$$
$$z :: \langle u[\hookrightarrow \mathrm{op}\{s\}[l;p;\triangleright;l']]\ ,\ u[\hookrightarrow \mathrm{op}\{s\}[l;p;\triangleright;l'']]\ ,\ a/v\ ,\ q\rangle$$

A matching process is terminated using rule 8 below. Note that this rule can only be applied when the matching process has been succesful, that is, when, after a number of applications of the rules 3–7 a potential redex $\overline{t}$ has been made equal to the lefthand side of a rule $\overline{t_1}$, and A has become the set of assignments such that $\overline{t_1}$ substituted by A is equal to $\overline{t}$.

$$z :: \langle \triangleright[\hookleftarrow p]\ ,\ \triangleright[\hookleftarrow p]\ ,\ a/v\ ,\ p_2[l_2]\rangle \xrightarrow{8} z :: \langle \triangleright[\hookrightarrow p_2[\triangleright;l_2]]\ ,\ a\rangle$$

An application of rule 8 changes the matching task into a replacement task. For that we have introduced in Σ_U the operation symbol '$_ :: \langle _\ ,\ _\rangle$'. The first argument of '$_ :: \langle _\ ,\ _\rangle$' is a pair formed by a rewrite theory an a term decomposed into a context an a redex; the second argument is the righthand side $\overline{t_2}$ of a rule whose lefthand side has been succesfully matched with the redex; the third argument is the set A of assignments resulting from this matching process. A replacement task is carried out as a two-phase process: the first phase consists in applying the substitution A to $\overline{t_2}$, and the second consists in the actual replacement using the result of this substitution.

The application of the substitution A to the term $\overline{t_2}$ is performed by rules 9–12 below and follows the same recursive style as the matching process.

$$z :: \langle u[\hookrightarrow \mathrm{op}\{s\}[l;\triangleright;q[l_1];l']]\ ,\ a\rangle \xrightarrow{9} z :: \langle u[\hookrightarrow \mathrm{op}\{s\}[l;\triangleright;l']][\hookrightarrow q[\triangleright;l_1]]\ ,\ a\rangle$$

$$z :: \langle u[\hookrightarrow \mathrm{var}\{x\}[\triangleright;\cdot]]\ ,\ (a;(p \mapsto \mathrm{var}\{x\}[\cdot]);a')\rangle \xrightarrow{10}$$
$$z : \langle u[\hookleftarrow p]\ ,\ (a;(p \mapsto \mathrm{var}\{x\}[\cdot]);a')\rangle$$

$$z :: \langle u[\hookrightarrow \mathrm{op}\{s\}[l;\triangleright]]\ ,\ a\rangle \xrightarrow{11} z :: \langle u[\hookleftarrow \mathrm{op}\{s\}[l]]\ ,\ a\rangle$$

$$z :: \langle u[\hookrightarrow \mathrm{op}\{s\}[l;\triangleright;l'][\hookleftarrow p]]\ ,\ a\rangle \xrightarrow{12} z :: \langle u[\hookrightarrow \mathrm{op}\{s\}[l;p;\triangleright;l']]\ ,\ a\rangle$$

Finally, the actual replacement is handled in the expected way by rule 14 below.

$$t\ddot{@}u[\hookrightarrow p] :: \langle \triangleright[\hookleftarrow p']\ ,\ a\rangle \xrightarrow{13} t@u[\hookrightarrow p']$$

3.3 Correctness of the universal theory

In this section we prove that the representation function $\overline{(_ \vdash _)}$ makes the theory U a $\mathcal{C}$-universal theory, for $\mathcal{C}$ the class of unconditional and unsorted finitely presentable rewrite theories. We also assume that for any rule $t \longrightarrow t'$ in $T \in \mathcal{C}$, $\mathrm{var}(t') \subseteq \mathrm{var}(t)$. Thus, the main result will be the following,

Theorem 3.2 *For any theory $T = (\Sigma, \emptyset, R) \in \mathcal{C}$, $t, t' \in T_\Sigma$,*

$$T \vdash t \longrightarrow t' \iff U \vdash \overline{T \vdash t \longrightarrow t'}.$$

This theorem will be proved by structural induction on rewriting logic proofs. Consider now the following rules of deduction:

2′. **S-Congruence** For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{t_i \longrightarrow t_i'}{f(t_1, ..., t_{i-1}, t_i, t_{i+1}, ..., t_n) \longrightarrow f(t_1, ..., t_{i-1}, t_i', t_{i+1}, ..., t_n)}$$

3′. **S-Replacement** For each rewrite rule $t(x_1, ..., x_n) \longrightarrow t'(x_1, ..., x_n)$ in T,

$$\frac{}{t(w_1/x_1, ..., w_n/x_n) \longrightarrow t'(w_1/x_1, ..., w_n/x_n)}$$

Lemma 3.3 *Any sequent derivable by the rules* 1–4, *can be derived by the rules* $\{1, 2', 3', 4\}$.

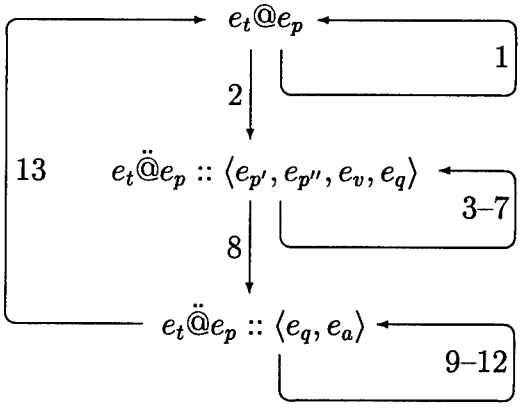
Given a theory $T \in \mathcal{C}$, a rewrite $t \longrightarrow t'$ is called a *one-step rewrite* if and only if it can be derived from T by finite application of the rules 1–2′–3′, with at least one application of the rule 3′. We say that a term is *irreducible* if and only if there does not exit any one-step rewrite $t \longrightarrow t'$. Finally, we say that a term $f(t_1, ..., t_n)$ is *only reducible at the top* if and only if t_i is irreducible for $1 \leq i \leq n$.

Lemma 3.4 *For any rule* $t(x_1, \ldots, x_n) \longrightarrow t'(x_1, \ldots, x_n)$ *in* U, *and any substitution* $w_1/x_1, \ldots, w_n/x_n$, *such that* w_i *is irreducible for* $1 \leq i \leq n$, *then* $t'(w_1/x_1, \ldots, w_n/x_n)$ *is only reducible at the top.*

As a corollary we can state the following lemma,

Lemma 3.5 *Any rewrite* $t \longrightarrow t'$ *derivable in* U *by the rules* 1–4, *and such that* t *is only reducible at the top, can be derived in* U *by the rules* $\{1, 3', 4\}$.

In particular, any rewrite $t \longrightarrow t'$ derivable in U by the rules $\{1, 3', 4\}$ will be an instance of a path in the following automaton, where a label l in an arc indicates a s-replacement rewrite with rule l. From now on, the metavariable e (possibly primed and/or subindexed) ranks over the set of irreducible terms in U_Σ.



In order to prove the direction ($\Rightarrow$) in Theorem 3.2 we prove first several technical lemmas.

For any terms e_u, e_v, we write $e_u \bigtriangledown e_v$ if and only if either $e_u = *$, or $e_u = e_{u'}[\rightarrow e]$, $e_v = e_{v'}[\rightarrow e]$, and $e_{u'} \bigtriangledown e_{v'}$. Also, for any terms e_u, e_v such that

$e_u \bigtriangledown e_v$,

$$e_v - e_u = \begin{cases} e_v & \text{if } e_u = * \\ e_{v'} - e_{u'} & \text{if } e_v = e_{v'}[\leftarrow e\,] \text{ and } e_u = e_{u'}[\leftarrow e\,]. \end{cases}$$

Lemma 3.6 *Given a rewrite* $\overline{T}@e_u[\rightarrow \overline{t}\,] \longrightarrow \overline{T}@e_{u'}[\rightarrow \overline{t'}\,]$, *then, for any term* e_v *such that* $e_u \bigtriangledown e_v$, *there is a rewrite* $\overline{T}@e_v[\rightarrow \overline{t}\,] \longrightarrow \overline{T}@e_{v'}[\rightarrow \overline{t'}\,]$ *such that* $e_{u'} \bigtriangledown e_{v'}$ *and* $e_v - e_u = e_{v'} - e_{u'}$.

Proof. By Lemma 3.5, $\overline{T}@e_u[\rightarrow \overline{t}\,] \longrightarrow \overline{T}@e_{u'}[\rightarrow \overline{t'}\,]$, is a composition of n rewrites of any of these forms,

(1) $\overline{T}@e_u[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};\overline{t_i};\overline{t_{i+1}};\ldots;\overline{t_n}\,]\,] \xrightarrow{+1}$
$\overline{T}@e_u[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};*;\overline{t_{i+1}};\ldots;\overline{t_n}\,]\,][\rightarrow \overline{t_i}\,]$,

(2) $\overline{T}@e_u[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};*;\overline{t_{i+1}};\ldots;\overline{t_n}\,]\,][\rightarrow \overline{t_i}\,] \xrightarrow{-1}$
$\overline{T}@e_u[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};\overline{t_i};\overline{t_{i+1}};\ldots;\overline{t_n}\,]\,]$, or

(3) $\overline{T}@e_u[\rightarrow \overline{t}\,] \xrightarrow{2;(3-7)^*;8;(9-12)^*;13} \overline{T}@e_u[\rightarrow \overline{t'}\,]$,

where, from now on, $+1$ and -1, indicate a s-replacement rewrite with rule 1 from left to right and from right to left, respectively; also, ";" indicates sequential composition of rewrites. It is easy to prove that given a rewrite $\overline{T}@e_u[\rightarrow \overline{t}\,] \longrightarrow \overline{T}@e_{u'}[\rightarrow \overline{t'}\,]$ of the form (1), (2) or (3), for any term e_v such that $e_u \bigtriangledown e_v$, there is a rewrite $\overline{T}@e_v[\rightarrow \overline{t}\,] \longrightarrow \overline{T}@e_{v'}[\rightarrow \overline{t'}\,]$ such that $e_{u'} \bigtriangledown e_{v'}$ and $e_v - e_u = e_{v'} - e_{u'}$. Then, applying transitivity $n-1$ times we obtain the desired result. □

Lemma 3.7 *For any theory* $T = (\Sigma, \emptyset, R)$ *in* $\mathcal{C}$, $f \in \Sigma_n$, $t_1, \ldots, t_n \in T_\Sigma$, *for any* $1 \leq i \leq n$, *if there is a rewrite* $\overline{T}@*[\rightarrow \overline{t_i}\,] \longrightarrow *[\rightarrow \overline{t'_i}\,]$, *then there is a rewrite*

$$\overline{T}\,@*[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};\overline{t_i};\overline{t_{i+1}};\ldots;\overline{t_n}\,]\,] \longrightarrow$$
$$\overline{T}\,@*[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};\overline{t'_i};\overline{t_{i+1}};\ldots;\overline{t_n}\,]\,].$$

Proof. The following is a rewrite in U,

$$\overline{T}\,@*[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};\overline{t_i};\overline{t_{i+1}};\ldots;\overline{t_n}\,]\,] \xrightarrow{+1}$$
$$\overline{T}\,@*[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};*;\overline{t_{i+1}};\ldots;\overline{t_n}\,]\,][\rightarrow \overline{t_i}\,] \xrightarrow{\delta}$$
$$\overline{T}\,@*[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};*;\overline{t_{i+1}};\ldots;\overline{t_n}\,]\,][\rightarrow \overline{t'_i}\,] \xrightarrow{-1}$$
$$\overline{T}\,@*[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};\overline{t'_i};\overline{t_{i+1}};\ldots;\overline{t_n}\,]\,],$$

where the rewrite δ exists by Lemma 3.6. □

From now on, we assume that for any set of assignments $A = (\vec{w}/\vec{x})$, $w_i \in T_\Sigma$, for $1 \leq i \leq n$. Also, given a set of assignments A, $\mathrm{var}(A)$ is the set of variables assigned in A. Finally, for any sets of assignments A, A', we write $A \Diamond A'$ if and only if for any terms $t \in T_\Sigma(X)$, $t' \in T_\Sigma$, if $t' = t(A')$, then $t' = t(A)(A')$. In particular, if $A \Diamond A'$, then for any assignment $a \in A'$, $A \cup \{a\} \Diamond A'$.

Lemma 3.8 *For any set of assignments A, and any set of variables V, given a rewrite*

$$e_z :: e_p, e_{p_1}, \overline{A}/\overline{V}, e_q\rangle \overset{(3-7)^*}{\longrightarrow} e_z :: e_{p'}, e_{p_2}, \overline{A'}/\overline{V'}, e_q\rangle,$$

then $A \subseteq A'$.

Lemma 3.9 *Given a theory $T = (\Sigma, \emptyset, R)$ in $\mathcal{C}$, $t \in T_\Sigma(X)$, $t' \in T_\Sigma$, and $t' = t(\vec{w}/\vec{x})$, then, for any set of assignments A, and any set of variables V, such that $\mathrm{var}(A) \cap V = \emptyset$, $(\vec{x}) \subseteq \mathrm{var}(A) \cup V$, and $A\Diamond(\vec{w}/\vec{x})$, there is a rewrite*

$$e_z :: \langle e_u[\hookrightarrow \triangleright(\overline{t})\,]\ ,\ e_u[\hookrightarrow \triangleright(\overline{t'})\,]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle \longrightarrow$$
$$e_z :: \langle e_u[\hookleftarrow \overline{t'}\,]\ ,\ e_u[\hookleftarrow \overline{t'}\,]\ ,\ \overline{A'}/\overline{V'}\ ,\ e_q\rangle$$

such that $t' = t(A')$, $\mathrm{var}(A') \cap V' = \emptyset$, $(\vec{x}) \subseteq \mathrm{var}(A') \cup V'$, and $A'\Diamond(\vec{w}/\vec{x})$, where from now on, for any $t \in T_\Sigma(X)$, such that $\overline{t} = e_p[e_l\,]$, $\triangleright(\overline{t}) = e_p[\triangleright, e_l\,]$.

Proof. By structural induction on t.

(1) $t = c$, for $c \in \Sigma_0$, $t' = c(\vec{w}/\vec{x})$. Then, the following is a rewrite in U

$$e_z :: \langle e_u[\hookrightarrow \triangleright(\overline{c})\,]\ ,\ e_u[\hookrightarrow \triangleright(\overline{c})\,]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle \overset{6}{\longrightarrow}$$
$$e_z :: \langle e_u[\hookleftarrow \overline{c}\,]\ ,\ e_u[\hookleftarrow \overline{c}\,]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle,$$

such that $c = c(A)$.

(2) $t = x_i$, $w_i = x_i(\vec{w}/\vec{x}) = w_i$. We have to consider two subcases:

(a) $x_i \in V$. Then, the following is a rewrite in U

$$e_z :: \langle e_u[\hookrightarrow \triangleright(\overline{x_i})\,]\ ,\ e_u[\hookrightarrow \triangleright(\overline{w_i})\,]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle \overset{4}{\longrightarrow}$$
$$e_z :: \langle e_u[\hookleftarrow \overline{w_i}\,]\ ,\ e_u[\hookleftarrow \overline{w_i}\,]\ ,\ \overline{(A \cup \{w_i/x_i\})}/\overline{(V - \{x_i\})}\ ,\ e_q\rangle,$$

such that $w_i = x_i(A \cup \{w_i/x_i\})$, $\mathrm{var}(A \cup \{w_i/x_i\}) \cap (V - \{x_i\}) = \emptyset$, $(\vec{x}) \subseteq \mathrm{var}(A \cup \{w_i/x_i\}) \cup (V - \{x_i\})$, and $A \cup \{w_i/x_i\}\Diamond(\vec{w}/\vec{x})$.

(b) $x_i \in \mathrm{var}(A)$. By assumption $A\Diamond(\vec{w}/\vec{x})$. Then, the following is a rewrite in U

$$e_z :: \langle e_u[\hookrightarrow \triangleright(\overline{x_i})\,]\ ,\ e_u[\hookrightarrow \triangleright(\overline{w_i})\,]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle \overset{5}{\longrightarrow}$$
$$e_z :: \langle e_u[\hookleftarrow \overline{w_i}\,]\ ,\ e_u[\hookleftarrow \overline{w_i}\,]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle,$$

with $w_i = x_i(A)$.

(3) $t = f(t_1, \ldots, t_n)$, $f \in \Sigma_n$, $n > 0$, $t' = f(t'_1, \ldots, t'_n)$, $t'_i = t_i(\vec{w}/\vec{x})$, for $1 \leq i \leq n$. Then, the following is a rewrite in U,

$$e_z :: \langle e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\triangleright; \overline{t_1}; \ldots; \overline{t_n}\,]\,]\ ,\ e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\triangleright, \overline{t'_1}; \ldots; \overline{t'_n}\,]\,]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle \overset{\beta}{\longrightarrow}$$
$$e_z :: \langle e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1}; \ldots; \overline{t'_n}; \triangleright\,]\,]\ ,\ e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1}; \ldots; \overline{t'_n}; \triangleright\,]\,]\ ,\ \overline{A'}/\overline{V'}\ ,\ e_q\rangle \overset{6}{\longrightarrow}$$
$$e_z :: \langle e_u[\hookleftarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1}; \ldots; \overline{t'_n}\,]\,]\ ,\ e_u[\hookleftarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1}; \ldots; \overline{t'_n}\,]\,]\ ,\ \overline{A'}/\overline{V'}\ ,\ e_q\rangle,$$

where the rewrite β is a composition of n rewrites of the form

$$e_z :: \langle\, e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1}; \ldots; \overline{t'_{i-1}}; \triangleright; \overline{t_i}; \ldots; \overline{t_n}\,]\,]\ ,$$
$$e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1}; \ldots; \overline{t'_{i-1}}; \triangleright; \overline{t'_i}; \ldots; \overline{t'_n}\,]\,]\ ,\ \overline{A_i}/\overline{V_i}\ ,\ e_q\rangle \overset{3}{\longrightarrow}$$
$$e_z :: \langle\, e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1}; \ldots; \overline{t'_{i-1}}; \triangleright; \ldots; \overline{t_n}\,]\,][\hookrightarrow \triangleright(\overline{t_i})\,]\ ,$$
$$e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1}; \ldots; \overline{t'_{i-1}}; \triangleright; \ldots; \overline{t'_n}\,]\,][\hookrightarrow \triangleright(\overline{t'_i})\,]\ ,\ \overline{A_i}/\overline{V_i}\ ,\ e_q\rangle \overset{\beta_i}{\longrightarrow}$$

$$e_z :: \langle e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1};\ldots;\overline{t'_{i-1}};\triangleright;\ldots;\overline{t_n}]][\leftarrow \overline{t'_i}]\ ,$$
$$e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1};\ldots;\overline{t'_{i-1}};\triangleright;\ldots;\overline{t'_n}]][\leftarrow \overline{t'_i}]\ ,\ \overline{A'_i/V'_i}\ ,\ e_q\rangle \xrightarrow{7}$$
$$e_z :: \langle e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1};\ldots;\overline{t'_{i-1}};\overline{t'_i};\triangleright;\ldots;\overline{t_n}]]\ ,$$
$$e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\overline{t'_1};\ldots;\overline{t'_{i-1}};\overline{t'_i};\triangleright;\ldots;\overline{t'_n}]]\ ,\ \overline{A'_i/V'_i}\ ,\ e_q\rangle,$$

where, by induction hypothesis, the rewrite β_i exists, for $1 \leq i \leq n$, and is such that $t'_i = t_i(A'_i)$, $\mathrm{var}(A'_i) \cap V'_i = \emptyset$, $(\vec{x}) \subseteq \mathrm{var}(A'_i) \cup V'_i$, $A'_i \Diamond (\vec{w}/\vec{x})$. Note also that by Lemma 3.8,

$$A = A_1 \subseteq A'_1 = A_2 \subseteq \ldots \subseteq A'_n = A'.$$

Therefore, $f(t'_1,\ldots,t'_n) = f(t_1,\ldots,t_n)(A')$, $\mathrm{var}(A') \cap V' = \emptyset$, $(\vec{x}) \subseteq \mathrm{var}(A') \cup V'$, and $A' \Diamond (\vec{w}/\vec{x})$.

□

Lemma 3.10 *Given a theory $T = (\Sigma, \emptyset, R)$ in $\mathcal{C}$, $t \in T_\Sigma(X)$, $t' \in T_\Sigma$, and $t' = t(\vec{w}/\vec{x})$, there is a rewrite*

$$e_z :: \langle e_u[\hookrightarrow \triangleright(\overline{t})]\ ,\ \overline{\vec{w}/\vec{x}}\rangle \longrightarrow e_z :: \langle e_u[\leftarrow \overline{t'}]\ ,\ \overline{\vec{w}/\vec{x}}\rangle,$$

Proof. By structural induction on t analogously to the proof of Lemma 3.9.□

Theorem 3.11 *For any theory $T = (\Sigma, \emptyset, R)$ in $\mathcal{C}$, $t, t' \in T_\Sigma$,*

$$T \vdash t \longrightarrow t' \implies U \vdash \overline{T}@*[\rightarrow \overline{t}] \longrightarrow \overline{T}@*[\rightarrow \overline{t'}]$$

Proof. By structural induction on rewriting logic proofs. For the reflexivity and transitivity rules, the result is obvious.

(1) S-Congruence. Given a rewrite in T

$$f(t_1,\ldots,t_{i-1},t_i,t_{i+1},\ldots,t_n) \longrightarrow f(t_1,\ldots,t_{i-1},t'_i,t_{i+1},\ldots,t_n),$$

with $t_i \longrightarrow t'_i$ as premise, by induction hypothesis there is a rewrite in U

$$\overline{T}@*[\rightarrow \overline{t_i}] \longrightarrow \overline{T}@*[\rightarrow \overline{t'_i}],$$

and, by Lemma 3.7, the following is also a rewrite in U

$$\overline{T}@*[\rightarrow \overline{f(t_1,\ldots,t_{i-1},t_i,t_{i+1},\ldots,t_n)}] \longrightarrow$$
$$\overline{T}@*[\rightarrow \overline{f(t_1,\ldots,t_{i-1},t'_i,t_{i+1},\ldots,t_n)}]$$

(2) S-Replacement. Given a rewrite $t_1 \longrightarrow t_2$ in T, such that $t(\vec{x}) \longrightarrow t'(\vec{x})$ is a rule in T, and $t_1 = t(\vec{w}/\vec{x})$, $t_2 = t'(\vec{w}/\vec{x})$, the following is a rewrite in U

$$\langle \overline{V}\ ,\ (\ldots;\overline{t} \Rightarrow \overline{t'};\ldots)\rangle@*[\rightarrow \overline{t_1}] \xrightarrow{2}$$
$$\langle \overline{V}\ ,\ (\ldots;\overline{t} \Rightarrow \overline{t'};\ldots)\rangle\ddot{@}*[\rightarrow \overline{t_1}] :: \langle \triangleright[\hookrightarrow \triangleright(\overline{t})]\ ,\ \triangleright[\hookrightarrow \triangleright(\overline{t_1})]\ ,\ \cdot/\overline{V}\ ,\ \overline{t'}\rangle \xrightarrow{\alpha}$$
$$\langle \overline{V}\ ,\ (\ldots;\overline{t} \Rightarrow \overline{t'};\ldots)\rangle\ddot{@}*[\rightarrow \overline{t_1}] :: \langle \triangleright[\leftarrow \overline{t_1}]\ ,\ \triangleright[\leftarrow \overline{t_1}]\ ,\ \overline{A'/V'}\ ,\ \overline{t'}\rangle \xrightarrow{8}$$
$$\langle \overline{V}\ ,\ (\ldots;\overline{t} \Rightarrow \overline{t'};\ldots)\rangle\ddot{@}*[\rightarrow \overline{t_1}] :: \langle \triangleright[\hookrightarrow \triangleright\overline{t'}]\ ,\ \overline{A'}\rangle \xrightarrow{\beta}$$
$$\langle \overline{V}\ ,\ (\ldots;\overline{t} \Rightarrow \overline{t'};\ldots)\rangle\ddot{@}*[\rightarrow \overline{t_1}] :: \langle \triangleright[\leftarrow \overline{t_2}]\ ,\ \overline{A'}\rangle \xrightarrow{13}$$
$$\langle \overline{V}\ ,\ (\ldots;\overline{t} \Rightarrow \overline{t'};\ldots)\rangle@*[\rightarrow \overline{t_2}]$$

where by Lemma 3.9 the rewrite α exists, and is such that $t(A) = t_1 = t(\vec{w}/\vec{x})$, and by Lemma 3.10 the rewrite β exists as well, since by as-

sumption on the class of theories we are considering, $\mathrm{var}(t') \subseteq \mathrm{var}(t)$, and therefore, $t(A) = t_1 = t(\vec{w}/\vec{x})$ implies $t'(A) = t_2 = t'(\vec{w}/\vec{x})$.

□

In order to prove the direction ($\Leftarrow$) in Theorem 3.2 we prove first several technical lemmas, regarding the correctness of the rules for selecting a subterm (Lemma 3.12), solving a matching problem with the lefthand side of a rule (Lemma 3.17), and performing the consequent substitution on the righthand side of the rule (Lemma 3.18). We define now a partial function ($\underline{e}$) that intuitively is the inverse function of ($\overline{e}$), but also gives a term from decomposed representation of it, obtained by s-replacement rewrites with rule 1. In particular,

$$\underline{e} = \begin{cases} t & \text{if } e = *[\rightarrow \overline{t}\,] \\ \underline{e_{u'}[\rightarrow \mathrm{op}\{\hat{f}\}[\ldots;\overline{t};\ldots]]} & \text{if } e = e_{u'}[\rightarrow \mathrm{op}\{\hat{f}\}[\ldots;*;\ldots]][\rightarrow \overline{t}\,] \end{cases}$$

Lemma 3.12 *For any theory $T = (\Sigma, \emptyset, R) \in \mathcal{C}$, $t, t' \in T_\Sigma$, given a rewrite*

$$\overline{T}@e_u[\rightarrow \overline{t'}\,] \xrightarrow{1^*} \overline{T}@e_{u'}[\rightarrow e\,],$$

if $\underline{e_u[\rightarrow \overline{t'}\,]} = t$, then $\underline{e_{u'}[\rightarrow e\,]} = t$.

Proof. By induction on the number of applications of rules $+1$ and -1. □

Lemma 3.13 *For any theory $T = (\Sigma, \emptyset, R) \in \mathcal{C}$, $t, t' \in T_\Sigma$, given a rewrite*

$$\overline{T}@e_u[\rightarrow \overline{t'}\,] \xrightarrow{1^*} \overline{T}@e'_u[\rightarrow e\,],$$

if $\underline{e_u[\rightarrow \overline{t'}\,]} = t$, then there is a rewrite

$$\overline{T}@e_u[\rightarrow \overline{t'}\,] \xrightarrow{-1^*} \overline{T}@*[\rightarrow \overline{t}\,] \xrightarrow{+1^*} \overline{T}@e'_u[\rightarrow e\,].$$

Lemma 3.14 *For any theory $T = (\Sigma, \emptyset, R) \in \mathcal{C}$, $t, t', t_1, t_2 \in T_\Sigma$, given a rewrite*

$$\overline{T}@e_u[\rightarrow \overline{t_1}\,] \longrightarrow \overline{T}@e_u[\rightarrow \overline{t_2}\,],$$

such that, $t = \underline{e_u[\rightarrow \overline{t_1}\,]}$, $t' = \underline{e_u[\rightarrow \overline{t_2}\,]}$, if $T \vdash t_1 \longrightarrow t_2$, then $T \vdash t \longrightarrow t'$.

Proof. By induction on the size of e_u, where $\mathrm{size}(*) = 1$, and $\mathrm{size}(e_{u'}[\rightarrow e\,]) = 1 + \mathrm{size}(e_{u'})$. For $\mathrm{size}(e_u) = 1$, the result is obvious. For $\mathrm{size}(e_u) = n + 1$, assume that the lemma holds for any $e_{u'}$ such that $\mathrm{size}(e_{u'}) = n$. Then, given a rewrite

$$\overline{T}\,@\,e_{u'}[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};*;\overline{t_{i+1}};\ldots;\overline{t_n}\,]][\rightarrow \overline{t_i}\,] \longrightarrow$$
$$\overline{T}\,@\,e_{u'}[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};*;\overline{t_{i+1}};\ldots;\overline{t_n}\,]][\rightarrow \overline{t'_i}\,],$$

note that

$$\underline{e_{u'}[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};*;\overline{t_{i+1}};\ldots;\overline{t_n}\,]][\rightarrow \overline{t_i}\,]} =$$
$$\underline{e_{u'}[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};\overline{t_i};\overline{t_{i+1}};\ldots;\overline{t_n}\,]]}, \text{ and}$$

$$\underline{e_{u'}[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};*;\overline{t_{i+1}};\ldots;\overline{t_n}\,]][\rightarrow \overline{t'_i}\,]} =$$
$$\underline{e_{u'}[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1};\ldots;\overline{t_{i-1}};\overline{t'_i};\overline{t_{i+1}};\ldots;\overline{t_n}\,]]}.$$

Now, by assumption, $T \vdash t_i \longrightarrow t_i'$. Thus, by s-congruence,

$$T \vdash f(t_1, ..., t_{i-1}, t_i, t_{i+1}, ..., t_n) \longrightarrow f(t_1, ..., t_{i-1}, t_i', t_{i+1}, ..., t_n),$$

and, finally, by induction hypothesis, we obtain the desired result

$$T \vdash \underline{e_{u'}[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1}; \ldots; \overline{t_{i-1}}; \overline{t_i}; \overline{t_{i+1}}; \ldots; \overline{t_n}]]} \longrightarrow$$
$$\underline{e_{u'}[\rightarrow \mathrm{op}\{\hat{f}\}[\overline{t_1}; \ldots; \overline{t_{i-1}}; \overline{t_i'}; \overline{t_{i+1}}; \ldots; \overline{t_n}]]}.$$

□

Next, we define a partial function $\mathrm{snm}(e)$ that intuitively gives the cardinality of the set of subterms not yet matched of a term that is being matched:

$$\mathrm{snm}(e) = \begin{cases} 0 & \text{if } e = \triangleright \\ |\,\mathrm{sbt}(e_{l'})\,| + \mathrm{snm}(e_u) & \text{if } e = e_u[\hookrightarrow e_p[e_l; \triangleright; e_{l'}]] \\ \mathrm{snm}(e_u) & \text{if } e = e_u[\hookleftarrow e_p[e_l]], \end{cases}$$

where $\mathrm{sbt}(e)$ is a function that gives the set of subterms of a list of terms.

Remark 3.15 *For any rewrite* $\alpha = e_z :: \langle e_{p_1}, e_p, e_v, e_q\rangle \overset{(3-7)^*}{\longrightarrow} e_z :: \langle e_{p_2}, e_{p'}, e_{v'}, e_q\rangle$, *note that* $\mathrm{snm}(e_{p_1}) \geq \mathrm{snm}(e_{p_2})$. *In particular, if the rewrite* α *contains at least one application of the rule* 3, *then* $\mathrm{snm}(e_{p_1}) > \mathrm{snm}(e_{p_2})$.

Lemma 3.16 *For any rewrite* $\alpha =$

$$e_z :: \langle e_u[\hookrightarrow e_{p_1}], e_u[\hookrightarrow e_p], e_v, e_q\rangle \overset{(3-7)^*}{\longrightarrow} e_z :: \langle e_u[\hookleftarrow e_{p'}], e_u[\hookleftarrow e_{p'}], e_{v'}\ , e_q\rangle$$

there does not exist any rewrite $\alpha' =$

$$e_z :: \langle e_u[\hookleftarrow e_{p'}], e_u[\hookleftarrow e_{p'}], e_{v'}\ , e_q\rangle \overset{(3-7)^*}{\longrightarrow} e_z :: \langle e_{u'}[\hookleftarrow e_{p''}], e_{u'}[\hookleftarrow e_{p''}], e_{v''}, e_q\rangle,$$

such that $e_{p'} \neq e_{p''}$, *and* $e_u = e_{u'}$.

Proof. Assume a rewrite α' such that $e_{p'} \neq e_{p''}$. By definition of the rules 3–7, we have to consider two cases,

(1) α' is a composition of s-replacement rewrites with rules 7 and 6 only. It is clear in this case that $e_u \neq e_{u'}$.

(2) α' is a composition of s-replacement rewrites 3–7 which contains at least one application of the rule 3. Then, $\mathrm{snm}(e_u[\hookleftarrow e_{p'}]) > \mathrm{snm}(e_{u'}[\hookleftarrow e_{p''}])$ by Remark 3.15. Therefore, $e_u \neq e_{u'}$.

□

Lemma 3.17 *Given a theory* $T = (\Sigma, \emptyset, R)$ *in* $\mathcal{C}$, $t \in T_\Sigma(X)$, $t' \in T_\Sigma$, *then, for any set of assignments* A, *and any set of variables* V, *such that* $\mathrm{var}(A) \cap V = \emptyset$, *if there is a rewrite* $\alpha =$

$$e_z :: \langle e_u[\hookrightarrow \triangleright(\overline{t})]\ ,\ e_u[\hookrightarrow \triangleright(\overline{t'})]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle \longrightarrow$$
$$e_z :: \langle e_u[\hookleftarrow e]\ ,\ e_u[\hookleftarrow e]\ ,\ \overline{A'}/\overline{V'}\ ,\ e_q\rangle,$$

then $\underline{e} = t' = t(A')$ *and* $\mathrm{var}(A') \cap V' = \emptyset$.

Proof.

By structural induction on t.

(1) $t = c$, for $c \in \Sigma_0$, $t' = c(\vec{w}/\vec{x})$. Then, by Lemma 3.16, and the definition of rules 3–7, any rewrite α will be a s-replacement rewrite with rule 6,

$$e_z :: \langle e_u[\hookrightarrow \triangleright(\overline{c})\,]\ ,\ e_u[\hookrightarrow \triangleright(\overline{c})\,]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle \xrightarrow{6}$$
$$e_z :: \langle e_u[\hookleftarrow \overline{c}\,]\ ,\ e_u[\hookleftarrow \overline{c}\,]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle,$$

with $c = c(A)$.

(2) $t = x_i$. We have to consider two subcases:

(a) $x_i \in V$. By Lemma 3.16, and the definition of rules 3–7, any rewrite α will be a s-replacement rewrite with rule 4,

$$e_z :: \langle e_u[\hookrightarrow \triangleright(\overline{x_i})\,]\ ,\ e_u[\hookrightarrow \triangleright(\overline{w_i})\,]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle \xrightarrow{4}$$
$$e_z :: \langle e_u[\hookleftarrow \overline{w_i}\,]\ ,\ e_u[\hookleftarrow \overline{w_i}\,]\ ,\ \overline{(A \cup \{w_i/x_i\})}/\overline{(V - \{x_i\})}\ ,\ e_q\rangle,$$

with $w_i = x_i(A \cup \{w_i/x_i\})$ and $\mathrm{var}(A \cup \{w_i/x_i\}) \cap (V - \{x_i\}) = \emptyset$.

(b) $x_i \in \mathrm{var}(A)$. By Lemma 3.16, and the definition of rules 3–7, any rewrite will be a s-replacement rewrite with rule 5,

$$e_z :: \langle e_u[\hookrightarrow \triangleright(\overline{x_i})\,]\ ,\ e_u[\hookrightarrow \triangleright(\overline{w_i})\,]\ ,\ (\ldots;\overline{w_i}/\overline{x_i};\ldots)/\overline{V}\ ,\ e_q\rangle \xrightarrow{5}$$
$$e_z :: \langle e_u[\hookleftarrow \overline{w_i}\,]\ ,\ e_u[\hookleftarrow \overline{w_i}\,]\ ,\ (\ldots;\overline{w_i}/\overline{x_i};\ldots)/\overline{V}\ ,\ e_q\rangle,$$

with $w_i = x_i(\ldots, w_i/x_i, \ldots)$.

(3) $t = f(t_1, \ldots, t_n)$, $f \in \Sigma_n$, $n > 0$, $t' = f(t'_1, \ldots, t'_n)$. By Lemma 3.16, and the definition of rules 3–7, any rewrite α will be a composition of rewrites whose last step is a s-replacement rewrite with rule 6,

$$e_z :: \langle e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\triangleright;\overline{t_1};\ldots;\overline{t_n}\,]\,]\ ,\ e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[\triangleright,\overline{t'_1};\ldots;\overline{t'_n}\,]\,]\ ,\ \overline{A}/\overline{V}\ ,\ e_q\rangle \xrightarrow{\beta}$$
$$e_z :: \langle e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_n;\triangleright\,]\,]\ ,\ e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_n;\triangleright\,]\,]\ ,\ \overline{A'}/\overline{V'}\ ,\ e_q\rangle \xrightarrow{6}$$
$$e_z :: \langle e_u[\hookleftarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_n\,]\,]\ ,\ e_u[\hookleftarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_n\,]\,]\ ,\ \overline{A'}/\overline{V'}\ ,\ e_q\rangle.$$

Moreover, the rewrite β will be a composition of n rewrites of the following form,

$$e_z :: \langle\, e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_{i-1};\triangleright;\overline{t_i};\ldots;\overline{t_n}\,]\,]\ ,$$
$$e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_{i-1};\triangleright;\overline{t'_i};\ldots;\overline{t'_n}\,]\,]\ ,\ \overline{A_i}/\overline{V_i}\ ,\ e_q\rangle \xrightarrow{3}$$
$$e_z :: \langle\, e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_{i-1};\triangleright;\ldots;\overline{t_n}\,]\,][\hookrightarrow \triangleright(\overline{t_i})\,]\ ,$$
$$e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_{i-1};\triangleright;\ldots;\overline{t'_n}\,]\,][\hookrightarrow \triangleright(\overline{t'_i})\,]\ ,\ \overline{A_i}/\overline{V_i}\ ,\ e_q\rangle \xrightarrow{\beta_i}$$
$$e_z :: \langle\, e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_{i-1};\triangleright;\ldots;\overline{t_n}\,]\,][\hookleftarrow e_i\,]\ ,$$
$$e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_{i-1};\triangleright;\ldots;\overline{t'_n}\,]\,][\hookleftarrow e_i\,]\ ,\ \overline{A'_i}/\overline{V'_i}\ ,\ e_q\rangle \xrightarrow{7}$$
$$e_z :: \langle\, e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_{i-1};e_i;\triangleright;\ldots;\overline{t_n}\,]\,]\ ,$$
$$e_u[\hookrightarrow \mathrm{op}\{\hat{f}\}[e_1;\ldots;e_{i-1};e_i;\triangleright;\ldots;\overline{t'_n}\,]\,]\ ,\ \overline{A'_i}/\overline{V'_i}\ ,\ e_q\rangle.$$

Note that, by induction hypothesis, for any rewrite β_i, $1 \leq i \leq n$, $\underline{e_i} = t'_i = t_i(A'_i)$ and $\mathrm{var}(A') \cap V' = \emptyset$. Also, by Lemma 3.8,

$$A = A_1 \subseteq A'_1 = A_2 \subseteq \ldots \subseteq A'_n = A'.$$

Therefore, $\underline{\mathrm{op}\{\hat{f}\}[e_1;\ldots;e_n\,]} = f(t'_1, \ldots, t'_n) = f(t_1, \ldots, t_n)(A')$, and $\mathrm{var}(A') \cap V' = \emptyset$.

□

Lemma 3.18 *Given a theory $T = (\Sigma, \emptyset, R)$ in $\mathcal{C}$, $t \in T_\Sigma(X)$, $t' \in T_\Sigma$, if there is a rewrite $\alpha =$*

$$e_z :: \langle e_u[\hookrightarrow \triangleright(\overline{t'})\,]\ ,\ \overline{A}\rangle \overset{(9-12)^*}{\longrightarrow} e_z :: \langle e_u[\hookleftarrow e\,]\ ,\ \overline{A}\rangle,$$

then $\underline{e'} = t'(A)$.

Proof. This proof is analogous to the proof of Lemma 3.17. □

Theorem 3.19 *For any theory $T \in \mathcal{C}$, $t, t' \in T_\Sigma$,*

$$U \vdash \overline{T}@*[\rightarrow \overline{t}\,] \longrightarrow \overline{T}@*[\rightarrow \overline{t'}\,] \implies T \vdash t \longrightarrow t'.$$

Proof. For $t = t'$ the result is obvious. For $t \neq t'$, by Lemma 3.5 and Lemma 3.13, $U \vdash \overline{T}@*[\rightarrow \overline{t}\,] \longrightarrow \overline{T}@*[\rightarrow \overline{t'}\,]$ can be obtained by composition of n paths of the following form

$$\langle \overline{V}\ ,\ e_r; (\overline{l} \Rightarrow \overline{r}); e_{r'}\rangle@*[\rightarrow e_1\,] \xrightarrow{1^*}$$
$$\langle \overline{V}\ ,\ e_r; (\overline{l} \Rightarrow \overline{r}); e_{r'}\rangle@e_{u'}[\rightarrow e'\,] \xrightarrow{2}$$
$$z :: \langle \triangleright[\hookrightarrow \triangleright(\overline{l})\,]\ ,\ \triangleright[\hookrightarrow \triangleright(e')\,]\ ,\ \cdot/\overline{V}\ ,\ \overline{r}\rangle \overset{(3-7)^*}{\longrightarrow}$$
$$z :: \langle \triangleright[\hookleftarrow e_m\,]\ ,\ \triangleright[\hookleftarrow e_m\,]\ ,\ \overline{A}/\overline{V'}\ ,\ \overline{r}\rangle \xrightarrow{8}$$
$$z :: \langle \triangleright[\hookrightarrow \triangleright\overline{r}\,]\ ,\ \overline{A}\rangle \overset{(9-12)^*}{\longrightarrow}$$
$$z :: \langle \triangleright[\hookleftarrow e''\,]\ ,\ \overline{A}\rangle \xrightarrow{13}$$
$$\langle \overline{V}\ ,\ e_r; (\overline{l} \Rightarrow \overline{r}); e_{r'}\rangle@e_{u'}[\rightarrow e''\,] \xrightarrow{1^*}$$
$$\langle \overline{V}\ ,\ e_r; (\overline{l} \Rightarrow \overline{r}); e_{r'}\rangle\ddot{@}*[\rightarrow e_2\,]$$

where $z = \langle \overline{V}\ ,\ e_r; (\overline{l} \Rightarrow \overline{r}); e_{r'}\rangle\ddot{@}e_{u'}[\rightarrow e'\,]$.

Note that for each one of these paths we have the following: assuming that $\underline{e_1} = t_i \in T_\Sigma$, then by Lemma 3.12 $\underline{e'} = t_{i_1} \in T_\Sigma$. Thus, by Lemma 3.17, $t_{i_1} = l(A')$, and by Lemma 3.18, $\underline{e''} = r(A) = t_{i_2} \in T_\Sigma$. Note then that $T \vdash t_{i_1} \longrightarrow t_{i_2}$ is a s-replacement rewrite in T with rule $l \longrightarrow r$, and therefore, by Lemma 3.14, $T \vdash \underline{e_{u'}[\rightarrow \overline{t_{i_1}}\,]} \longrightarrow \underline{e_{u'}[\rightarrow \overline{t_{i_2}}\,]}$ also holds. Finally, note that by Lemma 3.12, $\underline{e_{u'}[\rightarrow \overline{t_{i_1}}\,]} = \underline{*[\rightarrow \overline{t_i}\,]}$ and $\underline{e_{u'}[\rightarrow \overline{t_{i_2}}\,]} = \underline{*[\rightarrow \overline{t'_i}\,]}$, for $t'_i \in T_\Sigma$. Therefore, applying transitivity $n-1$ times in T we obtain the desired rewrite. □

Therefore, Theorem 3.2 is proved by Theorem 3.11 and Theorem 3.19. □

4 Strategies in General Logics

A metacircular interpreter may have a fixed strategy, and such a strategy may remain at the metalevel. This will make such an interpreter less flexible, and will complicate formal reasoning about its correctness. Even if quite flexible strategies can be defined, say in a theorem-prover's tactic language, such a language may remain a programming language external to the logic that it is controlling. In such a situation, "control" becomes an extralogical feature of the system.

If strategies can be defined *inside* the logic that they control, we are in a much better situation, since formal reasoning within the system can be applied to the strategies themselves to prove important properties about them. Reflective logics offer good opportunities for defining internal strategy languages of this kind, because the metalevel being controlled can be expressed within the logic.

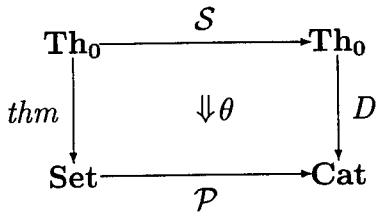
We give here general axioms for a strategy language, and for one internal to a logic. Again, we cover the case of entailment systems. Similar axioms for strategy languages at the proof calculus level can be found in [6].

Given a logical theory T, a strategy is a computational way of looking for certain theorems of T. This may be done by having a strategy language $\mathcal{S}(T)$ associated to T. Generally, we can think abstractly of the strategy language $\mathcal{S}(T)$ as a computational system in which strategy expressions can be further and further evaluated—in some cases perhaps forever, and sometimes in highly nondeterministic ways—so that the more we evaluate one such expression the more theorems will be *exhibited* in further stages. We can naturally represent such evaluations from a strategy expression E to another E' by labelled transitions $\alpha : E \longrightarrow E'$. If we denote by $\theta(E)$ the set of theorems exhibited by E, then our requirement is that if there is a computation $\alpha : E \longrightarrow E'$, then $\theta(E) \subseteq \theta(E')$. Of course, transitions $\alpha : E \longrightarrow E'$ and $\beta : E' \longrightarrow E''$ should be composable, to yield $\alpha;\beta : E \longrightarrow E''$, and it is always possible to add idle transitions $E : E \longrightarrow E$. Therefore, we can axiomatize the computations of the strategy language as a category $\mathcal{S}(T)$.

Definition 4.1 Given an entailment system $\mathcal{E}$, an *external strategy language* for it is a functor $\mathcal{S} : \mathbf{Th}_0 \longrightarrow \mathbf{Cat}$, together with a natural transformation $\theta : \mathcal{S} \Rightarrow \mathcal{P}_{fin} \circ thm$, where $\mathcal{P} : \mathbf{Set} \longrightarrow \mathbf{Cat}$ is the functor sending each set X to the poset $\mathcal{P}(X)$ of subsets of X, viewed as a category with arrows the subset inclusions. In addition, the strategy language is required to be *complete* in the sense that for each theorem $\varphi \in thm(T)$ there is an $E \in \mathcal{S}(T)$ such that $p \in \theta(E)$.

An *internal strategy language* for $\mathcal{E}$ consists of:

- an endofunctor $\mathcal{S} : \mathbf{Th}_0 \longrightarrow \mathbf{Th}_0$—that should be thought of as associating to each theory T a "local metatheory" $\mathcal{S}(T)$, axiomatizing strategies for T inside the logic;
- a functor $D : \mathbf{Th}_0 \longrightarrow \mathbf{Cat}$ assigning to each theory T a "category of deductions" $D(T)$, and a functor $U : \mathbf{Cat} \longrightarrow \mathbf{Set}$ such that $thm = U \circ D$; that is, from the deductions in $D(T)$ we can systematically obtain all the theorems of T using U;
- a natural transformation $\theta : D \circ \mathcal{S} \Rightarrow \mathcal{P} \circ thm$ such that $(D \circ \mathcal{S}, \theta)$ is an external strategy language for $\mathcal{E}$. □



We say that a strategy expression $E \in \mathcal{S}(T)$ is *Church-Rosser* iff whenever we have transitions $\alpha : E \longrightarrow E'$ and $\beta : E \longrightarrow E''$ there is always an E''' and transitions $\alpha' : E' \longrightarrow E'''$ and $\beta' : E'' \longrightarrow E'''$. We say that a strategy expression E is *sequential* iff there is a (finite or countable) sequence of nonidle transitions $\alpha_n : E_n \longrightarrow E_{n+1}$ with $E_0 = E$, such that for each nonidle transition $\alpha : E \longrightarrow E'$ there exists a unique k such that $\alpha = \alpha_0; \ldots; \alpha_k$. Each sequential strategy expression is obviously Church-Rosser, but the contrary is not true in general. Strategy expressions need not be Church-Rosser. For example, a strategy language can have a nondeterministic choice operator $_ \oplus _$ so that an expression $E \oplus E'$ has transitions $\alpha : E \oplus E' \longrightarrow E$, and $\beta : E \oplus E' \longrightarrow E'$. Such a nondeterministic choice operator should then be "opaque," so that no proofs are exhibited until it has disappeared, that is, $\theta(E \oplus E') = \emptyset$.

If a reflective entailment system has an internal strategy language, then the strategies $\mathcal{S}(U)$ for the universal theory are particularly important, since they represent at the object level strategies for computing in the universal theory. A *metacircular interpreter* for such a language can then be regarded as the implementation of a particular strategy in $\mathcal{S}(U)$. In general, it is easier to define internal strategy languages when a logic is reflective; and such languages can then be very expressive and powerful, not only for U, but also for all other theories. This is indeed the case for rewriting logic, as we discuss in the next section.

The usefulness of internal strategies is of course very general. For example, in the context of typed lambda calculi, the important advantages of having an internal strategy language has been stressed by several authors. Thus, using reflective capabilities both tactics and decision pocedures can be specified, reasoned about, and executed inside the Nuprl constructive type theory [1,7]. Similarly, Rueß[18] discusses in detail an elegant approach for endowing the calculus of constructions with internal strategies, as part of his treatment of reflection for such a calculus.

5 Strategies in Rewriting Logic

In this section we apply to rewriting logic the general semantic axiomas for internal strategy language presented in Section 4. We give a general method for defining and proving correct a strategy language, and show how the correctness of a universal theory greatly simplifies the proof of correctness of a given strategy language. Since we have proved rewriting logic reflective for the class $\mathcal{C}$ of unconditional and unsorted finitely presentable rewrite theories, we will define an internal strategy language for controlling the rewriting inference process of theories in $\mathcal{C}$.

As defined in Section 4 an internal strategy language is given by an endofunctor $\mathcal{S} : \mathbf{Th_0} \longrightarrow \mathbf{Th_0}$, satisfying additional semantic requirement; for our purposes $\mathbf{Th_0}$ will be the category of finitely presentable theories in rewriting logic but with morphisms restricted to identities. This endofuctor $\mathcal{S}$ associates to each rewrite theory R a "local metatheory" $\mathcal{S}(\mathcal{R})$, axiomatizing strategies for R in the logic.

Our idea is to use the reflective capabilities of rewriting logic, and in particular, the existence of a universal theory U, to define a subfunctor $\mathcal{K} \hookrightarrow \mathcal{S}$, what we call the *kernel* of the internal strategy language, whose correctness is based on the correctness of U itself as a universal theory. Then, a wide variety of endofunctors $\mathcal{S}$ can extend such a kernel axiomatizing additional strategies, but their correctness can be reduced to that of the kernel.

For example a typical kernel $\mathcal{K}$ can be defined as the following function on rewrite theories $R = (\Sigma, \emptyset, L, R)$ [3]

$$\mathcal{K}(\Sigma, \emptyset, \mathcal{L}, \mathcal{R}) = (\Sigma_{\mathcal{U}} \cup \{\mathrm{metapply}(_,_)\}, \mathrm{E_U} \cup \{\mathrm{def_l}\}_{\mathrm{l} \in \mathrm{L}}, \mathrm{L_U}, \mathrm{R_U}),$$

where if $l : t(\vec{x}) \longrightarrow t'\vec{x})$ in R, then $\mathrm{def_l}$ is the equation,

$$\mathrm{metapply}(\overline{\mathrm{t_1}}, \mathrm{l}) = \langle \cdot\ ,\ \cdot \rangle \ddot{@} * [\hookrightarrow \overline{\mathrm{t_1}}] :: \langle \triangleright[\hookrightarrow \triangleright(\overline{\mathrm{t}})]\ ,\ \triangleright[\hookrightarrow \triangleright(\overline{\mathrm{t_1}})]\ ,\ \cdot/\vec{\mathrm{x}}\ ,\ \overline{\mathrm{t'}} \rangle.$$

Now we can define $\mathcal{S}$ as extending $\mathcal{K}$, in the sense that it axiomatizes additional strategies, but always based on the *kernel* defined by $\mathcal{K}$. Consider a very simple endofunctor $\mathcal{S}(\Sigma, \emptyset, \mathcal{L}, \mathcal{R})$ that creates a theory with a signature $\Sigma_{\mathcal{S}} = \{\Sigma_U \cup \{\mathrm{metapply}(_,_), \mathrm{rw}_ \Rightarrow _\mathrm{with}_, \mathrm{failure}, \mathrm{idle}, \mathrm{apply}(_), _;_, \mathrm{result}(_), \mathrm{andthen}(_,_), \mathrm{error}*\}$, a set of equations $E_{\mathcal{S}} = \{E_U, \mathrm{def_l}\}$, and a set of rules $R_{\mathcal{S}} =$

$\mathrm{rew}\ \overline{\mathrm{t}} \Rightarrow \overline{\mathrm{t'}}\ \mathrm{with}\ \mathrm{apply(l)} \longrightarrow \mathrm{rew}\ \overline{\mathrm{t}} \Rightarrow \mathrm{result}(\mathrm{metapply}(\overline{\mathrm{t'}}, \mathrm{l}))\ \mathrm{with\ idle},$
$\mathrm{rew}\ \overline{\mathrm{t}} \Rightarrow \overline{\mathrm{t'}}\ \mathrm{with\ S;S'} \longrightarrow (\mathrm{rew}\ \overline{\mathrm{t}} \Rightarrow \overline{\mathrm{t'}}\ \mathrm{with\ S}) \longrightarrow \mathrm{andthen\ S'},$
$\mathrm{result}(*[\hookrightarrow \overline{\mathrm{t'}}]) \longrightarrow \overline{\mathrm{t'}},$
$\mathrm{result}(\langle \cdot\ ,\ \cdot \rangle \ddot{@} * [\hookrightarrow \overline{\mathrm{t}}] :: \langle \mathrm{u}[\hookrightarrow \mathrm{p[l]}]\ ,\ \mathrm{u}[\hookrightarrow \mathrm{q[l']}]\ ,\ \mathrm{a/v}\ ,\ \overline{\mathrm{t'}} \rangle) \longrightarrow \mathrm{error}*\ \mathrm{if\ p} \neq \mathrm{q},$
$(\mathrm{rew}\ \overline{\mathrm{t}} \Rightarrow \overline{\mathrm{t'}}\ \mathrm{with\ idle})\ \mathrm{andthen\ S'} \longrightarrow \mathrm{rew}\ \overline{\mathrm{t}} \Rightarrow \overline{\mathrm{t'}}\ \mathrm{with\ S'},$
$(\mathrm{failure})\ \mathrm{andthen\ S'} \longrightarrow \mathrm{failure})$

In this case it is trivial to define the natural transformation θ required by the definition of a strategy language that extracts from a rewriting strategy expression, the rewrites that it was supposed to describe

$$\theta(\mathrm{rew}\ \overline{\mathrm{t}} \Rightarrow \overline{\mathrm{t'}}\ \mathrm{with\ idle}) = \overline{\mathrm{t}} \Rightarrow \overline{\mathrm{t'}}.$$

Note that the correctness of this strategy language is based on the correctness of its kernel. In particular, the correctness of the apply(_) strategy depends on the correctness of the metapply(_,_) function—in conection with the result(_) function—in representing the replacement rule in rewriting logic. But this is a result that can be easily obtained as a corollary of Theorem 3.19. A more developed example of an internal strategy language based in reflection can be found in [4]

[3] As in Section 3.3—to simplify the exposition— we asume that the equations have transformed into bidirectional rules.

6 Concluding Remarks

We have specified a universal theory for rewriting logic and have proved its correctness. We have also introduced the notion of internal strategy language and have given a general method for defining such languages in rewriting logic. In joint work with Steven Eker and Patrick Lincoln we are applying these ideas and techniques in the context of the Maude language. Future developments and applications that we think particularly important include:

- Detailed construction and proof of a universal theory for the variant of rewriting logic whose underlying equational logic is membership equational logic [19,2].
- Applications of rewriting logic to give semantics to other reflective sytems and languages.
- Uses of reflection in logical framework applications of rewriting logic (see [5] for a discussion of this topic).
- Metaprogramming uses of reflection in rewriting logic, including general module composition and transformation operations [4], and special topics such as supercompilation [28].
- Further development of, and experimentation with, internal strategy languages.
- Development of formal environment extending rewriting logic languages.

Acknowledgement

We cordially thank Carolyn Talcott for her very helpful suggestions and her encouragement of this work along many discussions, and Narciso Martí-Oliet who read thorough many drafts of this paper and gave us most valuable comments and suggestions for improving the exposition. Our warm thanks also to our fellow members in the Maude team, Steven Eker and Patrick Lincoln, with whom we have exchanged many ideas about these topics, and with whom we have worked to realize them in the Maude language.

References

[1] William E. Aitken, Robert L. Constable, and Judith L. Underwood. Metalogical frameworks II: Using reflected decision procedures. Technical Report, Computer Sci. Dept., Cornell University, 1993; also, lecture at the Max Planck Institut für Informatik, Saarbrücken, Germany, July 1993.

[2] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. Manuscript, SRI International, August 1996.

[3] R. S. Boyer and J Strother Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In Robert Boyer and

J Moore, editors, *The Correctness Problem in Computer Science*, pages 103–185. Academic Press, 1981.

[4] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[5] Manuel G. Clavel and José Meseguer. Reflection and strategies in rewriting logic. This volume.

[6] Manuel G. Clavel and José Meseguer. Axiomatizing reflective logics and languages. In Gregor Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.

[7] Robert L. Constable. Using reflection to explain and enhance type theory. In Helmut Schwichtenberg, editor, *Proof and Computation*, volume 139 of *Computer and System Sciences*, pages 109–144. Springer, 1995.

[8] Fausto Giunchiglia, Paolo Traverso, Alessandro Cimatti, and Paolo Pecchiari. A system for multi-level reasoning. In *IMSA'92*, pages 190–195. Information-Technology Promotion Agency, Japan, 1992.

[9] John Harrison. Metatheory and reflection in theorem proving: a survey and critique. University of Cambridge Computer Laboratory, 1995.

[10] Patricia Hill and John Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[11] Douglas J. Howe. Reflecting the semantics of reflected proof. In Peter Aczel, Harold Simmons, and Stanley S. Wainer, editors, *Proof Theory*, pages 229–250. Cambridge University Press, 1990.

[12] Gregor Kiczales, editor. *Proceedings of Reflection'96, San Francisco, California, April 1996*. Xerox PARC, 1996.

[13] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[14] Narciso Martí-Oliet and José Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a Logical System?*, pages 355–392. Oxford University Press, 1994.

[15] Satoshi Matsuoka, Takuo Watanabe, Yuuji Ichisugi, and Akinori Yonezawa. Object-oriented concurrent reflective architectures. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing*, pages 211–226. Springer LNCS 612, 1992.

[16] Seán Matthews. Reflection in logical systems. In *IMSA'92*, pages 178–183. Information-Technology Promotion Agency, Japan, 1992.

[17] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, August 1995.

[18] Harald Rueß. Reflection of formal tactics in a deductive reflection framework. Manuscript, Universität Ulm, Abt. Künstliche Intelligenz, January 96.

[19] José Meseguer. Membership algebra. Lecture at the Dagstuhl Seminar on "Specification and Semantics," July 9, 1996. Extended version in preparation.

[20] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[21] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[22] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *IMSA'92*, pages 36–47. Information-Technology Promotion Agency, Japan, 1992.

[23] Luis H. Rodriguez, Jr. A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler. In *IMSA'92*, pages 107–112. Information-Technology Promotion Agency, Japan, 1992.

[24] N. Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge University Press, 1994.

[25] Brian Smith and Akinori Yonezawa, editors. *Proc. of the IMSA'92 International Workshop on Reflection and Meta-Level Architecture, Tokyo, November 1992.* Research Institute of Software Engineering, 1992.

[26] Brian C. Smith. Reflection and Semantics in Lisp. In *Proc. POPL'84*, pages 23–35. ACM, 1984.

[27] G. L. Steele, Jr. and G. J. Sussman. The art of the interpreter or, the modularity complex. Technical Report AIM-453, MIT AI-Lab, May 1978.

[28] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[29] M. Wand and D.P. Friedman. The mystery of the tower revealed. *Lisp and Symbolic Computation*, 1(1):11–38, 1988.

[30] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

Electronic Notes in Theoretical Computer Science 4 (1996)

A reflective extension of ELAN

Hélène Kirchner, Pierre-Etienne Moreau

INRIA Lorraine & CRIN-CNRS
615, rue du Jardin Botanique, BP 101
54602 Villers-lès-Nancy Cedex, France
http://www.loria.fr/equipe/protheo.html
email: Helene.Kirchner@loria.fr,moreau@loria.fr

Abstract

The expressivity of rewriting logic as meta-logic has been already convincingly illustrated. The goal of this paper is to explore the reflective capabilities of ELAN, a language based on the concepts of computational systems and rewriting logic. We define a universal theory for the class of ELAN programs and the representation function associated to this universal theory. Then we detail the effective transformations to implement and propose the definition of two built-in modules that provide the last step to get the reflective capabilities we want for the ELAN system.

1 Introduction

The expressivity of rewriting logic as a meta-logic able to encode various object logics, has been already convincingly illustrated in [Mes92,MOM93,KKV95a]. Reflective properties of rewriting logic are studied for instance in [CM96], in which metalogical axioms for reflective logics and declarative languages in general are proposed. The two notions of *universal theory* for a class of representable theories and of *representation function* are defined. In this approach the logic of choice is a parameter and the formalism is illustrated by the specific case of rewriting logic.

Our interest on reflection comes from the development of a declarative rewriting logic language called ELAN and the goal of this paper is to explore the reflective capabilities of ELAN.

ELAN is an environment dedicated to prototype, experiment and study the combination of different deduction systems for constraint solving, theorem proving and logic programming paradigms. The language is based on the concept of computational systems [KKV95a,Vit94,BKK+96b] given by a signature providing the syntax, a set of conditional rewrite rules describing the deduction mechanism, and a strategy to guide application of rewrite rules. Formally, this is a rewrite theory in rewriting logic [Mes92], [MOM93], together with a notion of strategy to select relevant computations. Strategy definitions in the currently distributed version of ELAN are based on a

© *1996 Elsevier Science B. V.*

small language combining labels of rewrite rules, a concatenation operator, an iterator `iterate`, and two selecting operators `dont know choose` and `dont care choose`, corresponding to non-deterministic and deterministic choices of strategies. Each ELAN module defines its own signature, labelled rewrite rules and strategies. ELAN is implemented in C++.

Four practical problems presented below motivate the design of a reflexive extension of ELAN.

Preprocessor: ELAN uses a pre-processing phase that allows describing the logic to be encoded in a flexible way. The pre-processor performs textual replacements and sometimes needs rewriting steps. So there are several interactions between the pre-processor and the abstract rewriting machine. In a reflective extension, the program written in the pre-processor syntax can be viewed as an ELAN term in an extended syntax, and can be transformed naturally by rewriting, using rewrite rules and strategies describing the pre-processing phase. Beyond the fact that this provides a unified semantics for the pre-processor and the interpretor, this also makes compilation of the system easier.

Strategy language: In automated deduction it is now a common approach to use a metalanguage to write strategies and tactics, specifying how object logic inference rules are composed to build proofs. We have developed in [BKK96a] a powerful strategy language for ELAN that is reflective in the sense that it is defined in rewriting logic. In order to implement it as an extension of the actual system, a systematic enrichment of the signatures, new rules and strategies must be added for describing strategy evaluation. In the first prototype of the strategy language, those systematic extensions are automatically done by ELAN, but this necessitated to modify the ELAN native code. A reflective extension of ELAN would allow prototyping the abstract strategy interpreter without any modification in the native code, as an extension of the computational systems.

Completion: In equational logic and equational programming languages, the Knuth-Bendix completion algorithm (or its variants) provides a way, when it succeeds, to transform an equational theory into a terminating and confluent set of rewrite rules with the same deductive power. Completion procedures, as many computational processes, can be formulated as instances of a schema that consists of applying rewrite rules on formulas with some strategy, until getting specific normal forms. In this sense they can be understood as computational systems. We have described in [KM95] completion algorithms in ELAN, in which rules to be completed are represented by terms, and the mechanism of simplification (ie. rewriting steps) is described in ELAN. In a reflective extension, we could transform terms, representing rules, into rewrite rules of the rewriting machine and use them to simplify the set of rules (represented by a set of terms) with the efficient ELAN rewriting mechanism.

Memorisation: Another possible use in ELAN of reflective capability would be for efficiency. Let $\mathcal{R}$ be a rewrite system, t a term and t' a normal form

of t w.r.t. $\mathcal{R}$. Let us assume that the normal form t' has to be computed several times; a reflective extension should permit adding the rewrite rule $t \Rightarrow t'$ and would transform $\mathcal{R}$ into a more efficient program.

In order to deal with program modification or extension, we can identify several problems to solve: representing (coding) programs by terms, simulating their execution, translating forth and back from programs to their representations, justifying the equivalence of deduction in both worlds of programs and their term representations.

In Section 2, we recall and adapt rewriting logic and universal theory to those rewrite theories which are actually used in ELAN with in particular conditions and local affectations in rewrite rules. In Section 3, we define an ELAN program that implements a universal theory for the class of ELAN programs and define the representation function associated to this universal theory. Then in Section 4, we detail the effective transformations to implement in order to make the system reflective. We propose the definition of two built-in modules that should be implemented in order to get the reflective capabilities we want for the ELAN system. Section 5 mentions some related work.

2 General setting

This section briefly presents the main concepts of general logic [Mes89] and rewriting logic [Mes92]. We slightly generalise the syntax and proof theory of conditional rewriting logic by introducing rules with local affectations. The notion of universal theory proposed by [CM96] is used to formalise metalogical axioms for reflective logics and languages.

2.1 Rewriting Logic

The definitions below are given in the unsorted case. The many-sorted and order-sorted cases can be handled in a similar, although more technical, manner. Our definitions are consistent with [DJ90,JK91] to which the reader is referred for more detailed considerations on universal algebra, term rewriting systems and matching.

We consider a set $\mathcal{F}$ of ranked function symbols, where $\mathcal{F}_n$ is the subset of functions of arity n, a set $\mathcal{X}$ of variables and the set of first-order terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ built on $\mathcal{F}$ and $\mathcal{X}$.

A *substitution* is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$, written $\sigma = \{y_1 \mapsto t_1, \ldots, y_k \mapsto t_k\}$. It uniquely extends to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We also use the notation $\{\overline{w} \mapsto \overline{x}\}(t)$ to express the simultaneous substitution of w_i for x_i in t. Letters $\sigma, \mu, \gamma, \phi, \ldots$ denote substitutions, and $\circ$ denotes their composition.

A $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equality is a pair of terms $\{t, t'\}$ in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, denoted as usual $(t = t')$. For any set of equalities E, $\mathcal{T}(\mathcal{F}, \mathcal{X})/E$ denotes the free quotient algebra of terms modulo E. The equivalence class of a term t modulo E is denoted $\langle t \rangle_E$ or just $\langle t \rangle$. For details and general results on calculus modulo

equational axioms, the reader is invited to consult for example [JK86]. To simplify notation, we denote a sequence of objects $(a_1, \ldots, a_n)$ by $\overline{a}$ or $\overline{a}^n$.

Syntax.

The syntax needed for defining a logic is provided by a signature which allows building sentences. In rewriting logic, a signature consists of a 3-tuple $\Sigma = (\mathcal{L}, \mathcal{F}, E)$, where $\mathcal{L}$ and $\mathcal{F}$ are sets of ranked function symbols and E is a set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities. Sentences built on a given signature are defined as sequents of the form $\pi \ : \ \langle t \rangle \rightarrow \langle t' \rangle$ where $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and π is the proof term representing the proof that allows to derive t' from t. So in rewriting logic, proofs are first-order objects identified with proof terms. In order to compose proofs, we introduce the infix binary operator "_;_" on proof terms. In order to record subproofs corresponding to conditions or local affectations, we introduce the operator "_{_}" whose second argument is a list of subproofs of the first argument. Therefore a proof term is by definition a term built on equivalence classes of $\mathcal{T}(\mathcal{F}, \mathcal{X})/E$, function symbols in $\mathcal{F}$, label symbols in $\mathcal{L}$, the composition operator "_;_", the subproof operator "_{_}". In other words, the set of proof terms is the term algebra $\mathcal{PT} = \mathcal{T}(\mathcal{L} \cup \{_;_\ ,\ _\{_\}\} \cup \mathcal{F} \cup \mathcal{T}(\mathcal{F}, \mathcal{X})/E)$. Lists of proof terms are built from the empty list "$\Box_{pt}$" and the concatenation operator "_ · _".

Since we need to be generic, we consider *Synt* a class of pairs (Σ, sen) consisting of a signature Σ together with a mapping *sen* associating to Σ the set of all legal sentences built on this signature.

Entailment systems.

For a given class of syntax *Synt* and (Σ, sen) in *Synt*, a theory T presented by a set of axioms Φ is the pair $T = (\Sigma, \Phi)$ where $\Phi \subseteq sen(\Sigma)$. Given a signature Σ, an entailment system is an abstract description of the provability relation of a sentence ϕ starting from a given set of sentences (also called axioms) Φ and using logical rules.

In rewriting logic, in order to build the entailment system, the notion of rewrite theory is introduced and an appropriate deduction system allows to inductively define the entailment relation.

A labelled rewrite theory is a 5-tuple $\mathcal{R} = (\mathcal{X}, \mathcal{F}, E, \mathcal{L}, R)$ where $\mathcal{X}$ is a given countably infinite set of variables, $\mathcal{L}$ and $\mathcal{F}$ are sets of ranked function symbols, E a set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities, and R is a set of rewrite rules denoted

$$\ell \ : \ \langle u \rangle \rightarrow \langle u' \rangle \ \ \textbf{if} \ \langle v_1 \rangle \rightarrow \langle v_1' \rangle \wedge \ldots \wedge \langle v_j \rangle \rightarrow \langle v_j' \rangle \\ \textbf{where} \ \sigma = \{y_1 \mapsto \langle a_1 \rangle, \ldots, y_k \mapsto \langle a_k \rangle\}$$

The part **if** $\langle v_1 \rangle \rightarrow \langle v_1' \rangle \wedge \ldots \wedge \langle v_j \rangle \rightarrow \langle v_j' \rangle$ is called the *condition* of the rule and **where** $\sigma = \{y_1 \mapsto \langle a_1 \rangle, \ldots, y_k \mapsto \langle a_k \rangle\}$ the *local affectation* of the rule. A rewrite rule may involve variables in its right-hand side that do not occur in its left-hand side, provided they are instantiated in the **where** part. To

indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occuring in either u or u', we write $u(x_1, \ldots, x_n) \to u'(x_1, \ldots, x_n)$.

A given labelled rewrite theory $\mathcal{R}$ entails the sequent $\pi \ : \ \langle t \rangle \to \langle t' \rangle$, which is denoted $\mathcal{R} \vdash \pi \ : \ \langle t \rangle \to \langle t' \rangle$, if $\pi \ : \ \langle t \rangle \to \langle t' \rangle$ is obtained by some finite application of the deduction rules in FIG. 1.

Reflexivity

$$t \ : \ \langle t \rangle \to \langle t \rangle$$

Congruence

$$\frac{\pi_i \ : \ \langle t_i \rangle \to \langle t_i' \rangle, \ i = 1 \ldots n}{f(\pi_1, \ldots, \pi_n) \ : \ \langle f(t_1, \ldots, t_n) \rangle \to \langle f(t_1', \ldots, t_n') \rangle}$$

Replacement. For each rewrite rule

[l] $\langle u(\overline{x}) \rangle \to \langle u'(\overline{x} \cup \overline{y}) \rangle$
if $\langle v_1(\overline{x} \cup \overline{y}) \rangle \to \langle v_1'(\overline{x} \cup \overline{y}) \rangle \wedge \ldots \wedge \langle v_j(\overline{x} \cup \overline{y}) \rangle \to \langle v_j'(\overline{x} \cup \overline{y}) \rangle$
where $\sigma = \{y_1 \mapsto a_1(\overline{x}), \ldots, y_k \mapsto a_k(\overline{x} \cup \overline{y}^{k-1})\}$

$$\frac{\begin{array}{rcl} \alpha_i & : & \langle w_i \rangle \to \langle w_i' \rangle, \ i = 1 \ldots n \\ \beta_1 & : & \langle a_1(\{\overline{x} \mapsto \overline{w}\}) \rangle \to \langle a_1'(\{\overline{x} \mapsto \overline{w}\}) \rangle \\ & \vdots & \\ \beta_k & : & \langle a_k(\{\overline{x} \cup \overline{y}^{k-1} \mapsto \overline{w} \cup \overline{a}^{k-1}\}) \rangle \to \langle a_k'(\{\overline{x} \cup \overline{y}^{k-1} \mapsto \overline{w} \cup \overline{a}^{k-1}\}) \rangle \\ \gamma_1 & : & \langle v_1(\{\overline{x} \cup \overline{y} \mapsto \overline{w} \cup \overline{a}\}) \rangle \to \langle v_1'(\{\overline{x} \cup \overline{y} \mapsto \overline{w} \cup \overline{a}\}) \rangle \\ & \vdots & \\ \gamma_j & : & \langle v_j(\{\overline{x} \cup \overline{y} \mapsto \overline{w} \cup \overline{a}\}) \rangle \to \langle v_j'(\{\overline{x} \cup \overline{y} \mapsto \overline{w} \cup \overline{a}\}) \rangle \end{array}}{\ell(\overline{\alpha}^n)\{\overline{\beta}^k.\overline{\gamma}^j\} \ : \ \langle u(\{\overline{x} \mapsto \overline{w}\}) \rangle \to \langle u'(\{\overline{x} \cup \overline{y} \mapsto \overline{w}' \cup \overline{a}'\}) \rangle}$$

Transitivity

$$\frac{\pi_1 \ : \ \langle t_1 \rangle \to \langle t_2 \rangle \quad \pi_2 \ : \ \langle t_2 \rangle \to \langle t_3 \rangle}{(\pi_1; \pi_2) \ : \ \langle t_1 \rangle \to \langle t_3 \rangle}$$

Fig. 1. Deduction rules for Rewriting Logic

The **Replacement** rule looks more complex than its version in [Mes92] for instance, due to the additional introduction of local affectations. Its hypotheses can be split into three parts: the first one (with proof terms α_i) for the deductions in the substitution part, the second one (with proof terms β_i) for the deductions in the local affectation part, and the third one (with proof terms γ_i) for the deductions in the condition part. But this is actually just a syntactical facility that could be handled in the same way as conditions in [Mes92]. Especially the proof term $\ell(\overline{\alpha}^n)\{\overline{\beta}^k.\overline{\gamma}^j\}$ is actually another notation for a proof term $\ell(\overline{\alpha}^n, \overline{\beta}^k, \overline{\gamma}^j)$ in the notation of [Mes92]. We prefer the more structured first notation that seems more readable to us. This corre-

spondence between the two notations allows us to directly reuse the results of [Mes92].

An equivalence on proof terms is defined by E and a set $E_{PT(R)}$ of equational axioms described in FIG. 2. This equivalence relation is important to relate different derivations with the same result, but different proofs.

$\forall \pi_1, \pi_2, \pi_3 \in PT \quad \pi_1; (\pi_2; \pi_3) = (\pi_1; \pi_2); \pi_3$	Associativity
$\forall \pi : \langle t \rangle \to \langle t' \rangle, \quad \pi; \langle t' \rangle = \pi, \text{ and } \langle t \rangle; \pi = \pi$	Local Identities
For all $f \in \mathcal{F}_n, n \in Nat, \quad \forall \pi_1, \ldots, \pi_n, \pi'_1, \ldots, \pi'_n$: $f(\pi_1; \pi'_1, \ldots, \pi_n; \pi'_n) = f(\pi_1, \ldots, \pi_n); f(\pi'_1, \ldots, \pi'_n)$	Independence
For all $f \in \mathcal{F}_n, n \in Nat$: $f(\langle t_1 \rangle, \ldots, \langle t_n \rangle) = \langle f(t_1, \ldots, t_n) \rangle$	Preservation of E
$\forall \ell : g \to d \in R, \forall \pi_1 : \langle t_1 \rangle \to \langle t'_1 \rangle, \ldots, \pi_n : \langle t_n \rangle \to \langle t'_n \rangle$ $\ell(\pi_1, \ldots, \pi_n) = \ell(\langle t_1 \rangle, \ldots, \langle t_n \rangle); d(\pi_1, \ldots, \pi_n)$ and $\ell(\pi_1, \ldots, \pi_n) = g(\pi_1, \ldots, \pi_n); \ell(\langle t'_1 \rangle, \ldots, \langle t'_n \rangle)$	Parallel Move Lemma

Fig. 2. $E_{PT(R)}$: Equivalence of proof terms

2.2 *Universal theory for Rewriting Logic*

A reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way. Two metatheoretic notions that can be reflected are theories and the entailment relation $\vdash$. This leads to the notion of universal theory, relative to a class C of *representable* theories proposed by [CM96].

For a given theory T, let $proofs(T)$ denote the set of all the proofs of theorems of the theory T.

Definition 2.1 Given an entailment system $\mathcal{E}$ and a class of theories C, a theory U is C-universal if there is a function, called *representation function*,

$$\overline{(_ \vdash _ : _)} : \cup_{T \in C} \{T\} \times proofs(T) \times sen(T) \longrightarrow sen(U)$$

such that for each $T \in C, p \in proofs(T), \varphi \in sen(T)$, $T \vdash p : \varphi \iff U \vdash \overline{T \vdash p : \varphi}$.

If, in addition, $U \in C$, then the entailment system $\mathcal{E}$ is called *C-reflective*.

Note that, if U is itself representable, we have a "reflective tower":

$$T \vdash p : \varphi \iff U \vdash q : \overline{T \vdash p : \varphi} \iff U \vdash q' : \overline{U \vdash q : \overline{T \vdash p : \varphi}}$$

In the appendix of [CM96], the authors give an example of a universal theory U for rewriting logic. Then, a representation function for U is defined with an overloaded symbol $\overline{(_)}$, described recursively from the representation

of theories, rewrite rules, terms, etc. In what follows we concentrate on the class C of finitely presentable conditional rewrite theories with local affectations that are the basis for ELAN. The problem is to find a theory U and a representation function making U a C-universal theory.

3 Universal Theory and ELAN

The notions of universal theory and representation function introduced in the previous section can be explained in terms of meta-level objects (such as signatures or sets of labelled rewrite rules and strategies) and base-level objects that are only terms. Using the terminology of [KSO95], we can call *meta-transformation* the representation function, that transforms meta-level objects into base-level objects, and *base-transformation* the converse transformation. In order to distinguish syntactically these two levels, different signatures are introduced. An ELAN program defines in a module $\mathcal{M}$ its own signature $Sig_{\mathcal{M}}$ and variables $Var_{\mathcal{M}}$, its set of rules expressed with terms in $\mathcal{T}(Sig_{\mathcal{M}}, Var_{\mathcal{M}})$ and strategies. An ELAN program can be translated into a term in an extended signature $Sig_{\mathcal{B}}$ using the representation function. Let $Mod_{\mathcal{B}}$ be an ELAN module that defines terms $\mathcal{T}(Sig_{\mathcal{B}}, Var_{\mathcal{B}})$ built on this signature $Sig_{\mathcal{B}}$ (close to the ELAN syntax for programs) and variables $Var_{\mathcal{B}}$.

The module $Mod_{\mathcal{B}}$ implements the universal theory U for rewriting logic and the considered class of ELAN rewrite theories. $Mod_{\mathcal{B}}$ is actually composed of several modules, mainly a module `msTerm` introducing signatures and many-sorted terms, `rwrule` building rewrite rules with conditions and local affectations, `proofterm` defining proof terms that record subproofs. Instead of giving extensively the module $Mod_{\mathcal{B}}$, we would rather illustrate a part of it focussing on the representation of many-sorted terms, rewrite systems and proof terms. Meanwhile we also illustrate below several features of the ELAN language more detailed in [KKV95a,KKV95b,BKK+96b].

3.1 Representing many-sorted terms

¿From a list of identifiers `Types`, the module `type` (see FIG. 3) builds base sorts (`type`) and a more complex functional type (`sig`). Here, the quantification `FOR EACH... SUCH THAT` means that identifiers are extracted from the list `Types` and declared as constant operators of sort `type`. `@` is the same as _ in the previous section.

This previous module `type`, combined with two lists of identifiers, is used by the module `msTerm` (see FIG. 4) to build operators and variables. More precisely, `msTerm` is parameterised by three lists: `Types` to import `type` with the correct parameter; `Vars`, a list of pairs of identifiers and types to define variables with their associated sorts; and `Ops`, a list of pairs of identifiers and signatures to define operators with their domains and codomains.

The line `F(@ {,@}~(N-1)) :({term}~N) term` in the module `msTerm` is expanded by the ELAN preprocessor into `F(@,...,@) : (term,...,term) term` according to the arity of `F`.

```
module type[Types]
sort type sig;
op global
        FOR EACH X:identifier SUCH THAT X:=(listExtract) elem(Types)
                :{ X : type; }
        @ --> @   : (list[type] type) sig
        dom(@)    : (sig) list[type];
        cod(@)    : (sig) type;
endop
...
end of module
```

Fig. 3. Types built from identifiers

```
module msTerm[Vars,Ops,Types]
import    global msTermCommons
op global
        FOR EACH X:pair[identifier,type]
        SUCH THAT X:=(listExtract)elem(Vars)
                :{ X          : variable; }
        FOR EACH F:pair[identifier,sig]
        SUCH THAT F:=(listExtract) elem(Ops) ANDIF arity(F)==0
                :{ F          : term;}
        FOR EACH F:pair[identifier,sig]; N:int
        SUCH THAT F:=(listExtract) elem(Ops) AND N:=() arity(F)
        ANDIF N > 0
                :{ F(@ {,@}^(N-1)) :({term}^N) term; }
endop
...
end of module
```

Fig. 4. Many-sorted terms

3.2 Representing rewrite systems and proof terms

FIG. 5 shows how to represent rewrite rules in ELAN; types, signatures and many-sorted terms are described in `msTerm`. Rewrite systems are represented as lists of rewrite rules. FIG. 6 illustrates the construction of proof terms.

```
module rwrule[Vars,Ops,Types,Labels]
import    global msTermCommons
                label[Labels];
sort rwrule rulebody;
op global
        @ => @                  : (term term) rulebody;
        @ if @                  : (rulebody term) rulebody;
        @ where @ := @          : (rulebody variable term) rulebody;
        [] @ end                : (rulebody) rwrule;
        [@] @ end               : (label rulebody) rwrule;
endop
...
end of module
```

Fig. 5. Rewrite rules in ELAN

Based on the signature defined by $Mod_{\mathcal{B}}$, the representation function can be defined for terms, rewrite rules and theories and proof terms. As in [CM96], we use an overloaded function symbol $\overline{(_)}$ to define the representation function.

- for R a set of rewrite rules $\{r_1, \ldots, r_n\}$, $\overline{R} = \overline{r_1}.\ldots.\overline{r_n}$;
  for an empty set R, $\overline{R} = \mathtt{nil}$

```
module proofterm[Vars,Ops,Types,Rwrules,Labels]
import    global    list[proofterm];
          local     msTermCommons
                    msApplySubstOn[term]
                    label[Labels];
sort proofterm;
op
 global
          @;@                  : (proofterm proofterm) proofterm;
          @(@)                 : (label substitution) proofterm;
          @[[@]]               : (proofterm list[proofterm]) proofterm;
endop
```

Fig. 6. Proof terms

- for r a rewrite rule $u \to u'$, $\overline{r} = \overline{u}$ => $\overline{u'}$.
  for r a rewrite rule $u \to u'$ **if** c **where** σ, $\overline{r} = \overline{u}$ => $\overline{u'}$ `if` $\overline{c}$ `where` $\overline{\sigma}$
- for t a term $f(t_1, \ldots, t_n)$, $f \in Sig_{\mathcal{M}}$, $n > 0$, $\overline{t}$ = `f(t_1,...,t_n)`
  where `f(@,...,@):(term ... term) term`,
  for t a term c (constant), $\overline{t}$ = `c` where `c : term`,
  for t a term x (variable), $\overline{t}$ = `x` where `x : variable`.
- for a proof term π,
  - if π is $\pi_1; \pi_2$, $\overline{\pi} = \overline{\pi_1}; \overline{\pi_2}$
  - if π is $\ell(\sigma)$, $\overline{\pi} = \overline{\ell}(\overline{\sigma})$
  - if π is $\pi'\{\lambda\}$, $\overline{\pi} = \overline{\pi'}[[\overline{\lambda}]]$

With this translation, we can already represent simple programs (i.e. without strategies) by terms. It remains then to represent the entailment relation $\vdash$, which is the goal of the next Section 3.3.

Example 3.1 Let us consider the following program and suppose that we want to transform this meta-level object into its representation. In the signature defined by $Mod_{\mathcal{B}}$, the rewrite system becomes a term of sort `list[rwrule]`. Precisely the list

```
[] elem(cons(e,l)) => e end .
[] elem(cons(e,l)) => choice where choice:=elem(l) if non_empty_list(l)
end .  nil.
```

```
module list
import element
sort element list
op global
          nil                  : list;
          cons(@,@)            : ( element list ) list;
endop
rules for element
declare     e,choice : element;  l : list;
bodies
 [] elem(cons(e,l)) => e                  end
 [] elem(cons(e,l)) => choice
          where choice:=elem(l)
          if non_empty_list(l)
 end
end of rules
end of module
```

3.3 Simulating rewriting

In order now to encode rewriting in ELAN, it is needed to describe all the steps of matching, substituting and replacement in an elementary rewrite step and how to iterate them. Moreover it is actually required to encode conditional term rewriting with a special kind of *local affectation* inside the rules, introduced by the key word **where**. These local affectations are memorised during the elementary rewrite step in a substitution.

The rewriting mechanism is described with rewrite rules, applying on tuples $\pi : (t, \omega, r, \sigma, \mu)$ where π is a proof-term, t is the term to be rewritten, ω a position in t ($\wedge$ is used when no position is specified), r a rewrite rule in a set R of rewrite rules, σ and μ are substitutions. Note that sequents appearing in each rule are decorated with appropriate proof terms.

The representation function for sequents is defined as a rewrite step on a term built with a ternary operator $_ \bullet _ : _$ taking as arguments a rewrite theory, a proof term and a term:

$$\overline{T \vdash \pi \ : \ t \to t'} = \overline{T} \bullet \overline{t} \ : \ \overline{t} \ \to \ \overline{T} \bullet (\overline{t}; \overline{\pi}) \ : \ \overline{t'}$$

We introduce a **one_step** rewrite rule, applying on such triple:

$$\textbf{one_step} \ T \bullet t : t \quad \to \quad T \bullet (t; \pi) : t'$$

where π, t', r such that $r \in R$, and

$$t : (t, \wedge, r, Id, Id) \to t; \pi : (t', \wedge, r, Id, Id)$$

This rule chooses a candidate rule r in R, performs an elementary step to rewrite the term t into t' and produces the associated proof term π. Let r be the candidate rule to rewrite the term t. Its left and right-hand sides are denoted respectively $lhs(r)$ and $rhs(r)$. There are four phases in the elementary rewrite step: find a position ω in t and a substitution σ, such that $\sigma(lhs(r)) = t_{|\omega}$, eliminate *conditional parts* by normalising conditions and compare them to $\top$, ($\top$ is the built-in boolean value *true* and is deeply connected to the implementation of conditions in rewrite rules), eliminate *local affectations* by recording them in substitution σ, and then apply the substitution and the replacement to obtain the resulting term $t[\sigma(rhs(r))]_\omega$. Before applying the replacement, phases 2 and 3 (elimination of *conditional parts* and *local affectations*) are iterated until no more condition or local affectation is left. We can express this algorithm with four rewrite rules **match**, **if_elim**, **where_elim** and **replace** detailed below and a simple strategy: (**match**; **if_elim***; **where_elim***; **replace**). The associated proof term is built in a deterministic way, during the computation. This provides a description of the proof closer to the actual execution in ELAN, since the proof term corresponds exactly now to the computation trace provided by the system.

The notation $proof(t \to t')$ is used to select the proof term in the sequent $\pi \ : \ \langle t \rangle \to \langle t' \rangle$. The function nf applied to a term t computes the normal form of t with respect to the whole set of rewrite rules R. Below we describe the rules that allow to compute an elementary rewrite step.

$$\textbf{match}\ \pi \ :\ (t, \wedge, r, Id, Id) \ \rightarrow\ \pi; (\ell(\sigma)\{\Box_{pt}\}) \ :\ (t, \omega, r, \sigma, \sigma)$$
$$\text{where } \omega, \sigma \text{ such that } \sigma(lhs(r)) = t_{|\omega}$$

Starting with a term t and a rewrite rule r, the **match** phase records a position ω in t and a substitution σ such that $\sigma(lhs(r)) = t_{|\omega}$. The rule first builds the proof term $\ell(\sigma)$ corresponding to the application of the rewrite rule r labelled by ℓ and instantiation of variables occuring in its left-hand side by values given by σ. $r(\sigma)\{\Box_{pt}\}$ is the proof term under construction. The list of subproofs issued from local affectations in this matching phase is initialised by the empty list denoted by $\Box_{pt}$.

$$\textbf{where_elim}\ \pi \ :\ (\pi'\{\lambda\}) : (t, \omega, r\ where\ v := t', \sigma, \mu) \ \rightarrow$$
$$\pi; (\pi'\{\lambda.proof(\mu(t') \rightarrow t'')\}) \ :\ (t, \omega, r, \sigma \circ \{v \mapsto t''\}, \mu \circ \{v \mapsto t'\})$$
$$\text{where } t'' = nf(\mu(t'))$$

This rule eliminates *one* local affectation. The rewrite rule r is transformed by removing the local affectation $v := t'$. The second substitution μ records this local affection while the first substitution σ records the normal form of t' w.r.t the rewrite system. The substitution σ will be useful to instantiate the right-hand side of r in the last **replace** phase.

Each time a local affectation is eliminated, μ records its initial form. So, instantiating a term by μ means eliminating new variables that do not occur in the left-hand side of r. To achieve the construction of the proof term associated to the application of r, the proof term associated to the sequent $\mu(t') \rightarrow t''$ (denoted by $proof(\mu(t') \rightarrow t'')$) is recorded in the list of subproof terms λ.

$$\textbf{if_elim}\ \pi; (\pi'\{\lambda\}) \ :\ (t, \omega, r\ if\ c, \sigma, \mu) \ \rightarrow$$
$$\pi; (\pi'\{\lambda.proof(\mu(c) \rightarrow \top)\}) \ :\ (t, \omega, r, \sigma, \mu)$$
$$\text{if } nf(\mu(t')) = \top$$

This rule eliminates *one* condition. The rewrite rule r is transformed by removing the condition $if\ c$. The application of **if_elim** rule has only a side effect, it builds the subproof term associated to the normalisation of $\mu(c)$ into $\top$: $proof(\mu(c) \rightarrow \top)$.

$$\textbf{replace}\ \pi; (\pi'\{\lambda\}) \ :\ (t, \omega, r, \sigma, \mu) \ \rightarrow$$
$$\pi; t[\pi'\{\lambda\}]_{\omega} \ :\ (t[\sigma(rhs(r))]_{\omega}, \wedge, r, Id, Id)$$

At this stage of the calculus, the substitution σ is able to instantiate each variable of the right-hand side of r by a ground term. The replacement is performed and produces the result term, as well as the resulting proof term. Several applications of **Congruence** are simulated by building the proof term

$t[\pi'\{\lambda\}]_\omega$ describing the *replacement* under the appropriate context t.

Soundness and completeness of this encoding is expressed in the two next propositions whose proofs are given in Appendix.

Proposition 3.2 *For any application of strategy (***match***;* **if_elim****;* **where_elim****;* **replace***, there exists a finite number of applications of rules* **Reflexivity**, **Congruence**, **Replacement** *and* **Transitivity** *building the same sequent and associated proof term.*

Proposition 3.3 *For any application of rules* **Reflexivity**, **Congruence**, **Replacement** *and* **Transitivity**, *there exists a finite number of applications of the strategy (***match***;* **if_elim****;* **where_elim****;* **replace***), building the same sequent and an* equivalent *associated proof term.*

Example 3.4 Let us consider different proofs of $g(f(a)) \to f(b)$ using the following rules: $r_1 \;:\; a \to b$, and $r_2 \;:\; g(x) \to x$. With rules **Reflexivity**, **Congruence**, **Replacement** and **Transitivity**, applied in different orders, one can derive three sequents with different proof terms:

$$r_2(f(r_1)) \;:\; g(f(a)) \to f(b)$$

$$r_2(f(a)) \;:\; g(f(a)) \to f(a)$$

$$\text{and } g(f(r_1)); r_2(f(b)) \;:\; g(f(a)) \to g(f(b))$$

But only the two last ones can be derived using the strategy (**match**; **if_elim***; **where_elim***; **replace**). Indeed this is not a problem, since according to the Parallel Move Lemma on proof terms, the first one is equivalent to the two others.

These rewrite rules **match**, **if_elim**, **where_elim** and **replace** can easily be implemented in ELAN. We give below the strategy `rewrite_state` corresponding to (**match**; **if_elim***; **where_elim***; **replace**) and the ELAN program corresponding to an elementary rewriting step. The rules `if-elim` and `where-elim` are calling a strategy `rewrite` that corresponds to the normalisation process with all rules in R. Its implementation in ELAN is also given below.

```
strategy rewrite_state
  dont care choose(match)
  repeat
    dont care choose(if_elim where_elim)
  endrepeat
  dont care choose(replace)
end of strategy
```

```
rules for rewrite_state
declare
        s,t,g,d,c,a             : term;
        v                       : variable;
        rw                      : rwrule;
        rb                      : rulebody;
        sigma                   : substitution;
        omega                   : list[int];
        mc                      : contrainte;
        lab                     : label;
        pi,pi2                  : proofterm;
        lpi                     : list[proofterm];
        couple                  : pair[term,proofterm];

bodies
 [match] [s,nil,lab,rb,identity]:pi =>
        [s,omega,lab,rb,sigma]:pi;lab(sigma)[[nil]]
                where sigma := () contrainte_to_subst(mc)
                where mc := (matchs) g =? s at omega
                where omega := (chooseOccurence) nvocc(s)
                where g := () left(rb)
 end

 [if_elim] [s,omega,lab,rb if c,sigma]:pi;pi2[[lpi]] =>
        [s,omega,lab,rb,sigma]:pi;pi2[[second(couple).lpi]]
                if first(couple)==true
                where couple:=(rewrite) [sigma(c),proofnil]
 end

 [where_elim] [s,omega,lab,rb where v:=a,sigma]:pi;pi2[[lpi]] =>
        [s,omega,lab,rb,sigma o v->t]:pi;pi2[[second(couple).lpi]]
                where t:=() first(couple)
                where couple:=(rewrite) [sigma(a),proofnil]
 end

 [replace] [s,omega,lab,rb,sigma]:pi;pi2[[lpi]] =>
        [s[sigma(d)] at omega,omega,lab,rb,identity]:pi;s[ pi2[[lpi]] ] at sigma
                where d := () right(rb)
                if no_conditional_no_local_affectation_rewrite_rule(rb)
 end
end of rules
```

```
rule rewrite for pair[term,proofterm]
declare
        s,t             : term;
        rs              : rewrite_state;
        rwr             : rwrule;
        pi,pi2          : proofterm;
body
 [s,pi] => [t,pi;pi2]
        where pi2:=() rewrite_state_to_proofterm(rs)
        where t :=() rewrite_state_to_term(rs)
        where rs :=(rewrite_state) rewrite(s,rwr)
        where rwr:= (listExtract) elem(rwrules)
end of rule

strategy rewrite
  repeat
    dont care choose(rewrite)
  endrepeat
end of strategy
```

4 Implementation

In order to provide now a suitable implementation of the representation function, we propose a translation in two steps. We distinguish between the representation of meta-level objects (described in $Mod_{\mathcal{B}}$) and a low-level translation mechanism, which should be implemented independently from $Mod_{\mathcal{B}}$. So the user can now choose his/her own representation of terms or rewrite rules in a flexible manner. More formally, we define the representation function as the composition $\varphi_2 \circ \varphi_1$ where $\varphi_1 : \mathcal{T}(Sig_{\mathcal{M}}, Var_{\mathcal{M}}) \mapsto \mathcal{T}(Sig_{\mathcal{B}'}, Var_{\mathcal{B}'})$ and $\varphi_2 : \mathcal{T}(Sig_{\mathcal{B}'}, Var_{\mathcal{B}'}) \mapsto \mathcal{T}(Sig_{\mathcal{B}}, Var_{\mathcal{B}})$, where $Sig_{\mathcal{B}'} \subseteq Sig_{\mathcal{B}}$ and $Var_{\mathcal{B}'} \subseteq Var_{\mathcal{B}}$. φ_1 encodes the translation mechanism that transforms any meta-level object into a fixed low-level representation (here we choose a list of identifiers as example), and φ_2 corresponds to the transformation of this low-level representation into user defined representation described by $Mod_{\mathcal{B}}$. FIG. 7 illustrates this idea.

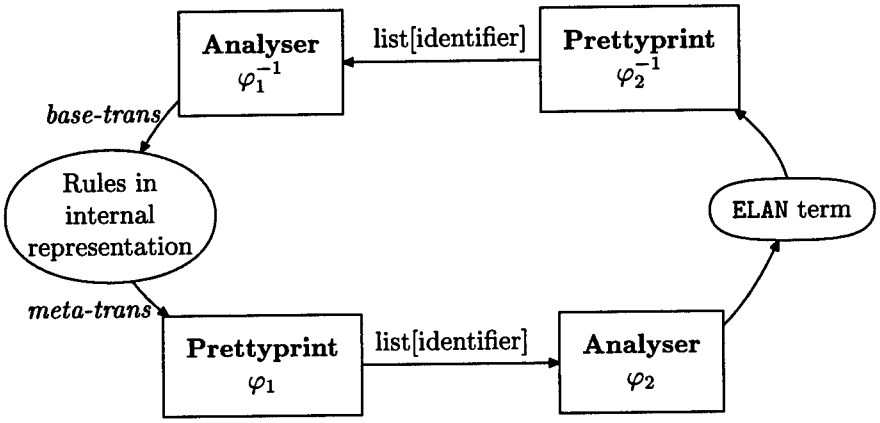


Fig. 7. Representation mechanism

The low-level representation consists only in identifiers, a binary concatenation operator '.' and a constant operator `nil` to terminate the list. Identifiers are built from strings of ASCII characters. In the implementation, it is needed to distinguish identifiers from operators '.' and `nil`, which compels us to introduce quotes '"'.

Instead of giving extensively the specification of φ_2 and φ_2^{-1}, we illustrate on an example what they do. Then we present two considered approaches to implement them.

Example 4.1 Given a term $t \in \mathcal{T}(Sig_{\mathcal{M}}, Var_{\mathcal{M}})$ such that $t = f(a, b)$. Its low-level representation is a term of $\mathcal{T}(Sig_{\mathcal{B}'}, Var_{\mathcal{B}'})$, namely the list of identifiers `op.{.f.}.[.a.b.].nil` (or more precisely `"op"."{"."f"."}"."["...`). An application of φ_2 on this representation transforms it into a term $t' \in \mathcal{T}(Sig_{\mathcal{B}}, Var_{\mathcal{B}})$: φ_2(`op.{.f.}.[.a.b.].nil`) = `f(a,b)`. φ_2^{-1} does the converse: φ_2^{-1}(`f(a,b)`) = `op.{.f.}.[.a.b.].nil`.

The first investigated approach consists in specifying φ_2 (resp. φ_2^{-1}) with rewrite rules. This amounts to describe a syntax analyser with rewrite rules that depend on the term representation described in $Mod_{\mathcal{B}}$. The following

piece of code illustrates an implementation of φ_2 for the representation adopted in Section 3, FIG. 4.

```
module phi2[Vars,Ops,Types]
op global
        phi2(@)                 : (list[identifier]) term;
        revphi2(@)              : (term) list[identifier];
endop
rules for term
declare
        F : identifier;
        L : list[identifier];
bodies
[]      phi2( "op" . "{" . F . "}" . "[" @ L @ "]" . nil ) =>
        F( FOR EACH X:identifier SUCH THAT X:=(listExtract) elem(L)
                :{ phi2(X), } )
end
[]      phi2( "op" . "{" . F . "}" . "[" . "--" . "]" . nil ) => F
end
end of rules

rules for list[identifier]
declare
        T : term;
bodies
[]      revphi2(T) =>
        "op" . "{" . head(T) . "}" . "[" .
                { revphi2(l-th subterm(T)) .}_l=1...arity(head(T)) "]" . nil
        if arity(head(T)) > 0
end
[]      revphi2(T) =>
        "op" . "{" . head(T) . "}" . "[" . "--" . "]" . nil
        if arity(head(T)) == 0
end
end of rules
```

Another solution to implement φ_2 would be to use the Earley algorithm implemented in ELAN to parse the low-level representation. Adopting this solution means that the low-level representation is no more accessible via `list[identifier]` and becomes an intermediate data structure immediately translated into a term of $\mathcal{T}(Sig_{\mathcal{B}}, Var_{\mathcal{B}})$ by a built-in function calling the Earley algorithm.

This presentation clearly shows the two built-ins φ_1 and φ_1^{-1} that have to be added to get a flexible reflective programming environment. φ_1 cannot be implemented with rewrite rules, because it has to access to information stored in C++ structures corresponding to meta-level objects. This mechanism has to be implemented in C++ and to be called as a *built-in* in ELAN programs. We think φ_1 is not really difficult to implement, since it is just a function that reads information in the memory. Conversely, φ_1^{-1} modifies the memory state and the ELAN programs. Implementing this functionality seems the most technical and difficult task to achieve in order to get interesting reflective capabilities.

5 Conclusion

In the literature, work on reflection covers a lot of different approaches.

Since in particular the seminal works of Gödel, Turing and Feferman, lo-

gicians have been mainly concerned with meta-reasoning, i.e. the ability to analyse inference steps or proofs in a given object logic using a second layer of logic called meta-logic. A meta-theory can be used to describe an object theory, to derive statements about the object theory, to control the search or to talk about provability in the object theory. Reflection up and down are two inference rules mentioned in [GT92] to formalise switching levels, and to theoretically justify for instance the combination of object-level and meta-level proofs and search in theorem proving. Logical frameworks and theorem provers, such as the systems FOL [Wey80,GS89] and GETFOL [Giu92,GT92], NuPRL [C+86,KC86], or ELF [Pfe94] to cite a few, are providing interesting meta-reasoning capabilities. A survey and critique on metatheory and reflection in theorem proving can be found in [Har95].

On the other hand, in computer science, reflection takes its roots in universal Turing machines and functions. Designers of programming languages have been most interested, as we are in this paper, by achieving, thanks to reflection, an extensible and flexible computation mechanism, and the ability to read and modify sources of programs. Several reflective languages have been designed in functional programming [Smi84], logic programming [BK82], object-oriented programming [Mae87] and rewrite system based languages [KSO95].

There is some hope of providing metalogical foundations for reflection in [CM96] that would unify many different approaches, and this is the reason why we adopted this view in our work. Complementary aspects of reflection in ELAN are mentioned in [BKK96a].

Acknowledgements

We sincerely thank Claude Kirchner and Peter Borovanský for helpful discussions and comments.

References

[BK82] K. Bowen and R. Kowalski. Amalgamating language and metalanguage in logic programming. In *Logic programming, APIC Studies in Data Processing*, volume 16, pages 153–172. Academic Press, 1982.

[BKK96a] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[BKK+96b] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1986.

[CM96] M. G. Clavel and J. Meseguer. Axiomatizing Reflective Logics and Languages. Proceedings of Reflection'96, pages 263–288. Xerox PARC, 1996.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[Giu92] F. Giunchiglia. The GETFOL manual-GETFOL version1. Technical Report 9204-01, IRST, Trento, Italy, 1992.

[GS89] F. Giunchiglia and A. Smail. Reflection in constructive and non-constructive automated reasoning. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, pages 123–140. MIT Press, 1989.

[GT92] F. Giunchiglia and P. Traverso. A metatheory of a mechanized object theory. Technical Report 9211-24, IRST, Trento, Italy, 1992.

[Har95] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique, 1995.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[KC86] T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Proceedings, Symposium on Logic in Computer Science*, pages 237–248, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.

[KKV95a] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, pages 131–158. The MIT press, 1995.

[KKV95b] C. Kirchner, H. Kirchner, and M. Vittek. ELAN *V 1.17 User Manual.* Inria Lorraine & Crin, Nancy (France), first edition, November 1995.

[KM95] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.

[KSO95] M. Kurihara, T. Sato, and A. Ohuchi. Reflective computation in term rewriting systems. *Computer Software*, 12(4):3–14, 1995.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In N. Meyrowitz, editor, *Proceedings of OOPSLA'87*, ACM, pages 147–155. ACM, 1987. Special issue of SIGPLAN Notices, Volume 22, Number 4.

[Mes89] J. Meseguer. General logics. In H.-D. E. et al., editor, *Logic Colloquium'87*, pages 275–329. Elsevier Science Publishers B. V. (North-Holland), 1989.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MOM93] N. Marti-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical report, SRI International, 1993.

[Pfe94] F. Pfenning. Elf: A meta-language for deductive systems (system description). In A. Bundy, editor, *12th International Conference on Automated Deduction*, LNAI 814, pages 811–815, Nancy, France, June 26–July 1, 1994. Springer-Verlag.

[Smi84] P. C. Smith. Reflection and semantics in lisp. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programmings Languages*, ACM, pages 23–35. ACM, 1984.

[Vit94] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, October 1994.

[Wey80] R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, (13):133–170, 1980.

6 Appendix

To improve readability, we denote by **sim** the strategy (**match**; **if_elim***; **where_elim***; **replace**). We also write $\pi \ : \ t \to t'$ instead of $\pi \ : \ \langle t \rangle \to \langle t' \rangle$.

Proposition 3.2.

For any application of the strategy **sim**, *there exists a finite number of applications of rules* **Reflexivity**, **Congruence**, **Replacement** *and* **Transitivity** *building the same sequent and associated proof term.*

Let r be a rewrite rule labelled by ℓ:

$\ell \ : \ u(\overline{x}) \to u'(\overline{x} \cup \overline{y})$
if $v_1(\overline{x} \cup \overline{y}) \to v_1'(\overline{x} \cup \overline{y}) \wedge \ldots \wedge v_j(\overline{x} \cup \overline{y}) \to v_j'(\overline{x} \cup \overline{y})$
where $\sigma = \{y_1 \mapsto a_1(\overline{x}), \ldots, y_k \mapsto a_k(\overline{x} \cup \overline{y}^{k-1})\}$

Without loss of generality, we can formulate r as:

$\ell \ : \ u(\overline{x}) \to u'(\overline{x} \cup \overline{y})$
if $v_1(\overline{x} \cup \overline{y}) \to \top \wedge \ldots \wedge v_j(\overline{x} \cup \overline{y}) \to \top$
where $\sigma = \{y_1 \mapsto a_1(\overline{x}), \ldots, y_k \mapsto a_k(\overline{x} \cup \overline{y}^{k-1})\}$

Let us prove Proposition 3.2 by induction on the depth n of the proof term resulting from the application of the strategy **sim**. The *depth* of a proof term is defined by:

- $\text{depth}(\pi_1;\pi_2) = \max(\text{depth}(\pi_1), \text{depth}(\pi_2))$
- $\text{depth}(t) = 0$
- $\text{depth}(f(\pi_1,\ldots,\pi_n)) = \max_{i=1,\ldots,n}(\text{depth}(\pi_i))$
- $\text{depth}(\ell(\overline{\alpha})\{\beta_1 \ldots \beta_k.\gamma_1 \ldots \gamma_j\}) =$
  $1 + \max(\max_{i=1,\ldots,k}(\text{depth}(\beta_i)), \max_{i=1,\ldots,j}(\text{depth}(\gamma_i)))$

▶ **case** $n = 1$: Then there is no condition or local affectation in r, $(j + k = 0)$. Applying rules **match** and **replace** on the tuple: $t : (t, \wedge, u(\overline{x}) \to u'(\overline{x}), Id, Id)$ yields:

$t : (t, \wedge, u(\overline{x}) \to u'(\overline{x}), Id, Id)$

$$\xrightarrow{\textbf{match}} t; \ell(\sigma)\{\Box_{pt}\} : (t, \omega, u(\overline{x}) \to u'(\overline{x}), \sigma, \sigma)$$
$$\xrightarrow{\textbf{replace}} t[\ell(\sigma)\{\Box_{pt}\}]_\omega : (t[\sigma(u'(\overline{x}))]_\omega, \wedge, u(\overline{x}) \to u'(\overline{x}), Id, Id)$$

Ground terms appearing in σ and their associated proofs terms $\overline{\alpha}$ are build by **Reflexivity**; an application of **Replacement** builds the sequent: $\ell(\overline{\alpha}) \ : \ t_{|\omega} \to u'(\{\overline{x} \mapsto \overline{w}\})$. Several applications of **Congruence** transform the sequent $\ell(\overline{\alpha}) \ : \ t_{|\omega} \to u'(\{\overline{x} \mapsto \overline{w}\})$ into $t[\ell(\overline{\alpha})]_\omega \ : \ t \to t[u'(\{\overline{x} \mapsto \overline{w}\})]_\omega$. The composition of **match** and **replace** corresponds to an application of the **Transitivity** deduction rule.

▶ **case** $n + 1$: Then there are n embedded applications of one-step rewriting in the evaluation of conditions and local affectations in r. Let us assume that we apply the rules **match**, m times $(m = j + k \geq 1)$ **where_elim** or **if_elim** and then **replace** on the tuple: $t : (t, \wedge, r, Id, Id)$ and get the result:

$$t; t[\ell(\sigma)\{\lambda\}]_\omega : (t[u'(\{\overline{x} \cup \overline{y} \mapsto \overline{a} \cup \overline{w}\})]_\omega, \wedge, u(\overline{x}) \to u'(\overline{x} \cup \overline{y}), Id, Id)$$

where $\lambda = \beta_1 \ldots \beta_k.\gamma_1 \ldots \gamma_j$ is a list of proof terms β_i and γ_i corresponding to evaluation of conditions and local affectations in r and of depth at most n.

By induction hypothesis applied to the proof terms in λ and the associated reductions, there is a finite number of applications of rules of rewriting logic that build these proof terms and sequents. An application of **Replacement** followed by several applications of **Congruence** then build the resulting sequent and proof term.

Proposition 3.3.

For any application of rules **Reflexivity**, **Congruence**, **Replacement** *and* **Transitivity**, *there exists a finite number of applications of the strategy* **sim** *building the same sequent and an* equivalent *associated proof term.*

Let prove by structural induction on proof term the expected result.
Suppose that sequents involved in premises of rules **Reflexivity**, **Transitivity**, **Congruence** and **Replacement** can be obtained in an *equivalent* form by applying **sim**. We want to prove that for each sequent produced by one application of those deduction rules, there exists a given sequence of applications of **sim** that produce an equivalent sequent.

▶ **Reflexivity:** This is trivially obtained by 0 application of the strategy **sim**.

▶ **Transitivity:** Let $\pi_1 \ : \ t_1 \to t_2$ and $\pi_2 \ : \ t_2 \to t_3$ be two sequents. By hypothesis, there exist a finite number of applications of **sim** that transforms the term t_1 (resp. t_2) into t_2 (resp. t_3) and produces the proof term π_1' (resp. π_2') equivalent to π_1 (resp. π_2). Successive applications of those two sequences of **sim** produce the proof term: $\pi_1'; \pi_2'$ which is equivalent to $\pi_1; \pi_2$.

▶ **Congruence:** Given $\pi_i \ : \ t_i \to t_i', \ i = 1 \ldots n$. An application of **Congruence** produces the sequent

$$f(\pi_1, \ldots, \pi_n) \ : \ f(t_1, \ldots, t_n) \to f(t_1', \ldots, t_n')$$

By induction hypothesis, equivalent sequents $\pi_i' \ : \ t_i \to t_i', \ i = 1 \ldots n$ can be produced by **sim**.

Starting from the term $f(t_1, \ldots, t_n)$, successive applications of **sim** produce:

$$f(\pi_1', t_2 \ldots, t_n); f(t_1', \pi_2', t_3 \ldots, t_n); \ldots; \ f(t_1', \ldots, t_{n-1}', \pi_n') \to \\ f(t_1, \ldots, t_n) \ : \ f(t_1', \ldots, t_n')$$

which is a proof term equivalent to $f(\pi_1', \ldots, \pi_n')$ so also to $f(\pi_1, \ldots, \pi_n)$. This equivalence uses repeatedly axioms of Independence and Local Identities.

▶ **Replacement:** Given $\alpha_i, i = 1 \ldots n$, $\beta_i, i = 1 \ldots k$ and $\gamma_i, i = 1 \ldots j$. By induction hypothesis, the list λ of subproof terms is equivalent to the list $\overline{\beta}^k.\overline{\gamma}^j$. An application of the (**match**, ..., **replace**) sequence at the top position ϵ produces the proof term $t[\ell(\sigma)\{\lambda\}]_\epsilon = \ell(\sigma)\{\lambda\}$ which is equivalent to $\ell(\overline{w}^n)\{\overline{\beta}^k.\overline{\gamma}^j\}$. Successive applications of **sim** will produce:

$$\ell(\overline{w}^n)\{\lambda\}; u'(\alpha_1', w_2, \ldots, w_n); u'(w_1', \alpha_2', w_3, \ldots, w_n); \ldots u'(w_1', \ldots, w_{n-1}', \alpha_n')$$

which is a proof term equivalent to $\ell(w_1, \ldots, w_n)\{\lambda\}; u'(\alpha_1', \ldots, \alpha_n')$ (thanks to Independence and Local Identities of FIG. 2).

Applying the axiom Parallel Move Lemma of FIG. 2, this proof term is equivalent to $\ell(\alpha_1', \ldots, \alpha_n')\{\lambda\}$ where the $\alpha_i', i = 1 \ldots n$ are equivalent to $\alpha_i, i = 1 \ldots n$. The resulting proof term is equivalent to $\ell(\overline{\alpha}^n)\{\overline{\beta}^k.\overline{\gamma}^j\}$.

Electronic Notes in Theoretical Computer Science 4 (1996)

Controlling Rewriting by Rewriting

Peter Borovanský, Claude Kirchner, Hélène Kirchner

INRIA Lorraine & CRIN-CNRS
615, rue du Jardin Botanique, BP 101
54602 Villers-lès-Nancy Cedex, France
`http://www.loria.fr/equipe/protheo.html`
email: `Peter.Borovansky,Claude.Kirchner,Helene.Kirchner@loria.fr`

Abstract

In this paper, we investigate the idea of controlling rewriting by strategies and we develop a strategy language whose operational semantics is also based on rewriting. This language is described in ELAN, a language based on computational systems that are simply rewriting theories controlled by strategies. We illustrate the syntax, semantics and different features of this strategy language. Finally, we sketch its bootstrapping implementation by a transformation into a computational system, whose heart is a rewrite theory controlled by a lower-level strategy of ELAN.

1 Introduction

Elegance and expressiveness of rewriting as a computational paradigm are no more to be stressed. What might be less evident, is the weakness that comes from the absence of controlling mechanism over rewriting. In many existing term rewriting systems, the term reduction strategy is hard-wired and is not accessible to the designer of an application. The results of [KKV95a] and some experiences show that even for medium size applications of rewriting logic, controlling rewriting becomes an important issue. The first successful implementation of a non-deterministic mechanism for controlling rewriting was implemented in C++ ([Vit94]) and later improved in [Vit96]. The idea of controlling rewriting is also investigated in Maude [CELM96].

This paper elaborates the idea of controlling rewriting *by rewriting*, more precisely by a strategy language based on rewriting. Roughly speaking, there are two levels of rewriting: the object (or first-order term) level, and the meta-level that controls the object-level. This idea was first mentioned in [GSHH92] and is more developed in this paper and its extended version [Bor96]. The main advance of our approach is that the controlling system is a rewrite system over typed proof terms that one-to-one correspond to computations at the object level. This approach is related to a view of strategies in reflective logics (in particular, rewriting logic) developed in [CM96]. ¿From this point of view, the strategy language described in this paper can be classified as an internal

© *1996 Elsevier Science B. V.*

strategy language, whose semantics and implementation are described in the same logic, while the language of elementary strategy available in the current distribution of ELAN and described in [Vit94] is an external one.

Another question which comes with the reflective approach is how to control computation at the meta-level. We first propose a solution in which the meta-level computation (i.e. the evaluation of strategies) is controlled by a built-in strategy. Later on, we show (not only for efficiency reason) that computation at the meta-level can be controlled by meta-strategies. Using a reflective logic allows using the same formalism at all levels, which might be viewed as an advantage of our approach. However, we stay realistic and we do not climb higher than the meta-meta-level.

In this paper, we first adopt the notion of strategy as subset of proof terms of a rewrite theory from [KKV95a,Vit94], which is recalled in Section 2. We define two subclasses of general strategies, called elementary and defined strategies whose operational semantics is given using rewriting logic. We concentrate in Section 3 on untyped elementary strategies, describe their operational semantics by a set of rewrite rules, their denotational semantics using sets of proof terms and show the correspondence between the two. Then in Section 4, we propose an axiomatisation in many-sorted rewriting logic of elementary and defined strategies. For that purpose, we outline a way of typing proof terms and strategies, then propose a declarative strategy language for user defined strategies also based on the paradigm of rewriting. In order to describe its implementation as a computational system in the ELAN system, we transform higher-level strategies defined in our framework into lower-level strategies of ELAN. The strategy language is illustrated first on small examples implemented in ELAN and Section 5 deals with a more complex example of a theorem prover in propositional logic and makes some comparisons with the 2OBJ system, before conclusion.

2 Proof terms and strategies

Rewriting logic is fully described in [Mes92,MOM93]. We briefly recall the basic notions and introduce notations used in the following. A *rewrite theory* [1] is a triple $\mathcal{RT} = (\Sigma, \mathcal{L}, \mathcal{R})$, where the signature Σ consists of sorts $\mathcal{S}$ and function symbols $\mathcal{F}$, $\mathcal{L}$ is a set of labels, and $\mathcal{R}$ is a set of rewrite rules. $\mathcal{R}$ can be defined as a subset of $\mathcal{L}(\mathcal{X}) \times \mathcal{T}(\mathcal{F} \cup \mathcal{X}) \times \mathcal{T}(\mathcal{F} \cup \mathcal{X})$, where $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$ stands for a set of terms built on function symbols $\mathcal{F}$ and variables $\mathcal{X}$. $\mathcal{L}(\mathcal{X})$ consists of linear and flat terms of the form $l(x_1, \ldots, x_n)$, where l is a rewrite rule label from $\mathcal{L}$ and $\{x_1, \ldots, x_n\}$ are pairwise different variables occurring in this rewrite rule. Here, we deal with rules $[l(\overline{x})] \quad u(\overline{x}) \Rightarrow u'(\overline{x})$ where $\overline{x} = \{x_1, \ldots, x_n\} = Var(u) \cup Var(u')$ and we do not suppose as usual that $Var(u') \subseteq Var(u)$.

Substitutions are assignments from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$, written $\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$, that uniquely extend to endomorphisms of $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$. We also use

[1] We restrict here for simplicity to many-sorted signatures without structural axioms.

the notation $t(\overline{w}/\overline{x})$ to express the simultaneous substitution of w_i for x_i in t.

The set of *(closed) proof terms* Π is defined as the set of terms $\mathcal{T}(\mathcal{F} \cup \mathcal{L} \cup \{;\})$ built on function symbols $\mathcal{F}$ of the rewrite theory, labels $\mathcal{L}$ and the symbol ';' standing for concatenation. Rewriting logic [Mes92,MOM93] consists of the first four axioms in Figure 1 and gives semantics to proof terms by interpretation of the symbol $_ : _ \Rightarrow _$ defined on $\Pi \times \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})$. In this context, $\pi : t \Rightarrow t'$ means there is a chain of rewrite steps from the term t to the term t' encoded by the proof term π. It is easy to see that for each proof term $\pi \in \Pi$, the sets

$$dom(\pi) = \{t \mid \exists\ t' \ s.t.\ \pi : t \Rightarrow t'\} \text{ and } cod(\pi) = \{t' \mid \exists\ t \ s.t.\ \pi : t \Rightarrow t'\}$$

are either both singletons or both empty. Thus, we can also define a partial function $_[_] : \Pi \times \mathcal{T}(\mathcal{F}) \mapsto \mathcal{T}(\mathcal{F})$, such that $\pi[dom(\pi)] = cod(\pi)$.

A *strategy* S is defined as a subset of Π. We extend the definition of the symbol $_ : _ \Rightarrow _$ for strategies as follows:

$$S : t \Rightarrow t' \text{ if there exists } \pi \in S, \text{ such that } \pi : t \Rightarrow t', \text{ and}$$
$$S[t] = \{t' \mid \pi \in S,\ \pi : t \Rightarrow t'\}.$$

The definition of domain can be extended too:

$$dom(S) = \{t \mid \exists t' \in \mathcal{T}(\mathcal{F}),\ \exists \pi \in S, \text{ s.t. } \pi : t \Rightarrow t'\}.$$

3 Elementary strategies

Clearly, an arbitrary strategy $S \subseteq \Pi$ may be very complicated or irregular (a non-recursive set) from the computational point of view. This is why we concentrate on languages describing special subclasses of strategies.

3.1 Rewriting logic of elementary strategies

The first step is a language of elementary strategies, where *an elementary strategy es* is an element of the set of terms

$$\mathcal{ES} = \mathcal{T}(\mathcal{F} \cup \mathcal{L} \cup \{;, \mathbf{id}, \mathbf{dc}, \mathbf{dk}, \mathbf{case}\})$$

Roughly speaking, elementary strategies represent non-recursive non-deterministic computations.

Let us start with the intuition behind elementary strategy symbols, which helps to understand the axioms in Figure 1. The strategy operator **dk** stands for *dont-known-choose*, **dc** for *dont-care-choose* and **case** is a 'sequential version' of dont-care-choose, which takes always the first, in textual order, successful branch[2]. The strategy constant **id** represents identity. $\mathbf{l}(es_1, \ldots, es_n)$ corresponds to an application of a rewrite rule labelled by $l \in \mathcal{L}$, which also applies substrategies es_i on values of variables x_i after matching and before replacing the matched subterm by the instantiated right-hand side of the rewrite rule. $\mathbf{f}(es_1, \ldots, es_n)$ is an application of substrategies es_i to subterms t_i of the term $f(t_1, \ldots, t_n)$ with root $f \in \mathcal{F}$.

[2] The **case** symbol is the *dont-care-choose* in the current version of ELAN. For details, see [KKV95b].

Reflexivity

$$t : t \Rightarrow t$$

Congruence

$$\frac{es_i : t_i \Rightarrow t'_i,\ i = 1..n}{\mathbf{f}(es_1, \ldots, es_n) : f(t_1, \ldots, t_n) \Rightarrow f(t'_1, \ldots, t'_n)}$$

Replacement by $[l(\overline{x})]\quad u(\overline{x}) \Rightarrow u'(\overline{x})$

$$\frac{es_i : w_i \Rightarrow w'_i,\ i = 1..n}{\mathbf{l}(es_1, \ldots, es_n) : \{\overline{x} \mapsto \overline{w}\}u \Rightarrow \{\overline{x} \mapsto \overline{w'}\}u'}$$

Transitivity

$$\frac{es_1 : t_1 \Rightarrow t_2,\quad es_2 : t_2 \Rightarrow t_3}{(es_1; es_2) : t_1 \Rightarrow t_3}$$

dk/dc

$$\frac{es_1 : t \Rightarrow t' \text{ or } es_2 : t \Rightarrow t'}{\mathbf{dk}(es_1, es_2) : t \Rightarrow t'} \qquad \frac{es_1 : t \Rightarrow t' \text{ or } es_2 : t \Rightarrow t'}{\mathbf{dc}(es_1, es_2) : t \Rightarrow t'}$$

case

$$\frac{es_1 : t \Rightarrow t' \text{ or } es_2 : t \Rightarrow t',\ t \notin dom(es_1)}{\mathbf{case}(es_1, es_2) : t \Rightarrow t'}$$

id

$$\mathbf{id} : t \Rightarrow t$$

Fig. 1. Rules for $\mathcal{ES}$ - definition of $es : t \Rightarrow t'$

In the rest of this paper, we deal only with labelled strategy terms $\mathbf{l}(es_1, \ldots, es_n)$ from $\mathcal{ES}$ for which there is a rewrite rule $[l(x_1, \ldots, x_n)]\quad u(\overline{x}) \Rightarrow u'(\overline{x})$ such that either $x_i \in Var(u) \cap Var(u')$ or $es_i \in \mathcal{T}(\mathcal{F})$, for any $i = 1..n$. This condition says that variables of a rewrite rule not occurring on both sides of a rule cannot be parameterised by an arbitrary proof term, only by a term from $\mathcal{T}(\mathcal{F})$. This condition allows us to determine a value for any variable even if it is not bound by matching. To define the formal semantics of these strategy symbols, we extend the set of deduction rules of rewriting logic by new rules for elementary strategies. The obtained system formalises the definition of the relation $es : t \Rightarrow t'$ (Figure 1), which is mutually recursive with the definition of the relation $t \in dom(es)$ (Figure 2). We can remark that the defined relation $es : t \Rightarrow t'$ makes no difference between **dk** and **dc** operators. Therefore, we have to define the operational semantics of elementary strategies in order to make precise the non-deterministic behaviour of **dc** and **dk**.

3.2 Operational semantics of elementary strategies

The description of operational semantics of elementary strategies is achieved through the definition of an interpreter described by labelled rewrite rules in Figure 3. The binary function symbol $_\lceil_\rceil$ stands for the non-deterministic application of elementary strategy es to the term t. Its evaluation is defined by the set of rewrite rules given in Figure 3. The result res of this application is a finite subset of terms. Moreover, this evaluation is not deterministic, due

Reflexivity-dom

$$t \in dom(t)$$

Congruence-dom

$$\frac{t_i \in dom(es_i), i = 1..n}{f(t_1, \ldots, t_n) \in dom(\mathbf{f}(es_1, \ldots, es_n))}$$

Replacement-dom by $[l(\overline{x})]$ $\quad u(\overline{x}) \Rightarrow u'(\overline{x})$

$$\frac{\theta x_i \in dom(es_i), i = 1..n}{t \in dom(\mathbf{l}(es_1, \ldots, es_n))} \quad \text{where } \theta(u) = t$$

Transitivity-dom

$$\frac{es_1 : t \Rightarrow t',\ t' \in dom(es_2)}{t \in dom(es_1; es_2)}$$

dk/dc-dom

$$\frac{t \in dom(es_1) \ \text{ or } \ t \in dom(es_2)}{t \in dom(\mathbf{dk}(es_1, es_2))} \qquad \frac{t \in dom(es_1) \ \text{ or } \ t \in dom(es_2)}{t \in dom(\mathbf{dc}(es_1, es_2))}$$

case-dom

$$\frac{t \in dom(es_1) \ \text{ or } \ t \in dom(es_2)}{t \in dom(\mathbf{case}(es_1, es_2))}$$

id-dom

$$t \in dom(\mathbf{id})$$

Fig. 2. Rules for $t \in dom(es)$

to **dc**, and we can obtain as many results as **dc**'s in the strategy *es*. For the representation of results, we use an abstract data type **List** defining lists with some pre-defined constructors like concatenation denoted by $.$, the empty list denoted by $\emptyset$, and operations like append denoted by $\cup$. More sophisticated operators are also defined over the type **List**, such as:

$$join_f(ws_1, \ldots, ws_n) = \{\mathbf{f}(e_1, \ldots, e_n) \mid e_i \in ws_i,\ i = 1 \ldots n\}, \text{ and}$$

$$join_t(ws_1, \ldots, ws_n) = \{\{x_i \mapsto e_i\}t \mid x_i \in Var(t),\ e_i \in ws_i,\ i = 1..n\},$$

The operator $_\lceil\lceil_\rceil\rceil$ is the generalisation of $_\lceil_\rceil$ on lists. All these operators are formally defined in Figure 4. The notation $res = (es\lceil t\rceil)_{\downarrow}$ means that *res* is *one of all* irreducible forms of the application $es\lceil t\rceil$, using the rules of the interpreter. Similarly, the notation $true = (t \in dom(es))_{\downarrow}$ means that there is a derivation of the term $t \in dom(es)$ terminating with *true*. Indeed the semantics of the symbol $_ \in dom(_)$ can be similarly described by a rewrite theory. The symbol $\ll^?$ stands for matching in the empty theory. A rewrite theory for matching is given in [JK91] and these rules should be added too to the interpreter. The following lemma links together the two symbols $_ \in dom(_)$ and $_\lceil_\rceil$.

Lemma 3.1 *(success and fail)*
If $true = (t \in dom(es))_{\downarrow}$ *then there exists a non-empty set* res *such that* $res = (es\lceil t\rceil)_{\downarrow}$. *If* $true \neq (t \in dom(es))_{\downarrow}$, *then* $res = (es\lceil t\rceil)_{\downarrow} = \emptyset$.

[DK]	$\mathbf{dk}(es_1, es_2)\lceil t\rceil$	$\Rightarrow es_1\lceil t\rceil \cup es_2\lceil t\rceil$
[DC_1]	$\mathbf{dc}(es_1, es_2)\lceil t\rceil$	$\Rightarrow res_1$ if $true = (t \in dom(es_1))_\downarrow$ and $\emptyset \neq res_1 = (es_1\lceil t\rceil)_\downarrow$
[DC_2]	$\mathbf{dc}(es_1, es_2)\lceil t\rceil$	$\Rightarrow res_2$ if $true = (t \in dom(es_2))_\downarrow$ and $\emptyset \neq res_2 = (es_2\lceil t\rceil)_\downarrow$
[DC_3]	$\mathbf{dc}(es_1, es_2)\lceil t\rceil$	$\Rightarrow \emptyset$ if $true \neq (t \in dom(es_1))_\downarrow$ and $true \neq (t \in dom(es_2))_\downarrow$
[$CASE_1$]	$\mathbf{case}(es_1, es_2)\lceil t\rceil$	$\Rightarrow res_1$ if $true = (t \in dom(es_1))_\downarrow$ and $\emptyset \neq res_1 = (es_1\lceil t\rceil)_\downarrow$
[$CASE_2$]	$\mathbf{case}(es_1, es_2)\lceil t\rceil$	$\Rightarrow es_2\lceil t\rceil$ if $true \neq (t \in dom(es_1))_\downarrow$
[ID]	$\mathbf{id}\lceil t\rceil$	$\Rightarrow \{t\}$
[$FSYM_1$]	$\mathbf{f}(es_1, \ldots, es_n)\lceil f(x_1, \ldots, x_n)\rceil$	$\Rightarrow join_f(es_1\lceil x_1\rceil, \ldots, es_n\lceil x_n\rceil)$
[$FSYM_2$]	$\mathbf{f}(es_1, \ldots, es_n)\lceil g(x_1, \ldots, x_m)\rceil$	$\Rightarrow \emptyset$ if $f \neq g$ or $m \neq n$
[LAB_1]	$\mathbf{l}(es_1, \ldots, es_n)\lceil u(x_1, \ldots, x_n)\rceil$	$\Rightarrow join_{u'}(\ldots, es_i\lceil x_i\rceil, \ldots, es_j, \ldots)$ where $[l(\overline{x})]\quad u(\overline{x}) \Rightarrow u'(\overline{x}) \in \mathcal{R}$, and $x_j \notin Var(u) \cap Var(u') \ni x_i$
[LAB_2]	$\mathbf{l}(es_1, \ldots, es_n)\lceil t\rceil$	$\Rightarrow \emptyset$ if $\emptyset = (u <<^? t)_\downarrow$ where $[l(\overline{x})]\quad u(\overline{x}) \Rightarrow u'(\overline{x}) \in \mathcal{R}$
[CONC]	$(es_1; es_2)\lceil t\rceil$	$\Rightarrow es_2\lceil\lceil\ es_1\lceil t\rceil\ \rceil\rceil$

Fig. 3. The interpreter of $\mathcal{ES}$ ($\mathcal{R}_\mathcal{I}$)

An example of an elementary strategy es, such that $\emptyset = (es\lceil t\rceil)_\downarrow$ and $true = (t \in dom(es))_\downarrow$ is given below.

Example 3.2 Considering labelled rules of the form $[ij]\quad i \Rightarrow j$, for all natural numbers i and j, one can check that $((\mathbf{dc}(01, 02)\ ;\ 11)\lceil 0\rceil)_\downarrow$ gives $\{1\}$ and $\emptyset$ as results. Indeed $0 \in dom(\mathbf{dc}(01, 02)\ ;\ 11)$.

The next result shows that the two views of operational semantics given in Figure 1 and Figure 3 relate together.

Lemma 3.3 *(operational semantics)*
If $es : t \Rightarrow t'$, then there exists a subset of ground terms $res \subseteq \mathcal{T}(\mathcal{F})$ such that $res = (es\lceil t\rceil)_\downarrow$ and $t' \in res$. Conversely if $t' \in res = (es\lceil t\rceil)_\downarrow$, then $es : t \Rightarrow t'$.

The interpreter described in Figure 3 models two levels of non-determinism. **dc**-non-determinism is modelled by the non-determinism of the rewrite theory and it is usually implemented by don't care heuristics which choose one out

$t \in dom(\mathbf{id}) \Rightarrow true$

$t \in dom(\mathbf{dk}(es_1, es_2)) \Rightarrow true$
if $true = (t \in dom(es_1))_{\downarrow}$ or $true = (t \in dom(es_2))_{\downarrow}$

$t \in dom(\mathbf{dc}(es_1, es_2)) \Rightarrow true$
if $true = (t \in dom(es_1))_{\downarrow}$ or $true = (t \in dom(es_2))_{\downarrow}$

$t \in dom(\mathbf{case}(es_1, es_2)) \Rightarrow true$
if $true = (t \in dom(es_1))_{\downarrow}$ or $true = (t \in dom(es_2))_{\downarrow}$

$f(t_1, \ldots, t_n) \in dom(\mathbf{f}(es_1, \ldots, es_n)) \Rightarrow true$
if $\forall i = 1..n,\ true = (t_i \in dom(es_i))_{\downarrow}$

$u(x_1, \ldots, x_n) \in dom(\mathbf{l}(es_1, \ldots, es_n)) \Rightarrow true$
if $\forall i = 1..n,\ true = (x_i \in dom(es_i))_{\downarrow}$
where $[l(\overline{x})]\ \ u(\overline{x}) \Rightarrow u'(\overline{x}) \in \mathcal{R}$

$t \in dom(es_1; es_2) \Rightarrow true$
if $true = (t \in dom(es_1))_{\downarrow}$ and $res = (es_1\lceil t \rceil)_{\downarrow}$ and $at_least_one(res, es_2)$

- at_least_one is defined by:

$at_least_one(a.as, es) \Rightarrow true$ if $true = (a \in dom(es))_{\downarrow}$

$at_least_one(a.as, es) \Rightarrow true$ if $true = at_least_one(as, es)_{\downarrow}$

- $join_{u'}$ is defined for any fixed $u' \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ by $2n + 1$ rewriting rules:

$For\ i = 1..n : join_{u'}(ws_1, \ldots, w_i.ws_i, \ldots, ws_n) \Rightarrow$
$join_{u'}(ws_1, \ldots, w_i, \ldots, ws_n) \cup join_{u'}(ws_1, \ldots, ws_i, \ldots, ws_n)$

$For\ i = 1..n : join_{u'}(ws_1, \ldots, \emptyset, \ldots, ws_n) \Rightarrow \emptyset$

$For\ 2n+1 : join_{u'}(w_1, \ldots, w_n) \Rightarrow \{x_i \mapsto w_i\}u'$

- $join_f$ is defined for any $f \in \mathcal{F}$ as follows:

$join_f(w_1, \ldots, w_n) \Rightarrow join_{f(x_1,\ldots,x_n)}(w_1, \ldots, w_n)$

- $_\lceil\lceil_\rceil\rceil$ is defined by:

$es\lceil\lceil a.as\rceil\rceil \Rightarrow es\lceil a\rceil \cup es\lceil\lceil as\rceil\rceil$

$es\lceil\lceil\emptyset\rceil\rceil \Rightarrow \emptyset$

Fig. 4. The interpreter - continuation of $\mathcal{ES}$

of many (usually the first) solutions. **dk**-non-determinism is modelled by collecting all particular results into a final set of results and it is usually implemented by using backtracking rather than handling sets of solutions.

Example 3.4 With the rewrite rules $[R_1]\ \ a \Rightarrow b$, $[R_2]\ \ a \Rightarrow c$, we get the results summarized in the next table, where the computations of the interpreter are represented by meta-level proof terms in the right column.

$es\lceil t\rceil$	$(es\lceil t\rceil)_{\downarrow}$	*Proof terms*
$\mathbf{dk}(R_1, R_2)\lceil a\rceil$	$\{b,\ c\}$	$\mathsf{DK}; \mathsf{LAB}_1; \mathsf{LAB}_1$
$\mathbf{dc}(R_1, R_2)\lceil a\rceil$	$\{b\}, \quad \{c\}$	$\mathsf{DC}_1; \mathsf{LAB}_1, \quad \mathsf{DC}_2; \mathsf{LAB}_1$
$\mathbf{case}(R_1, R_2)\lceil a\rceil$	$\{b\}$	$\mathsf{CASE}_1; \mathsf{LAB}_1$

Up-to-now we do not restrict interpreter computations by defining acceptable meta-proof terms. However, we will show later that reasoning about meta-strategies may be useful and may influence operational semantics and efficiency of the interpreter.

3.3 Denotational semantics of elementary strategies

The difference between **dc** and **dk** is now explained by introducing their denotational semantics. It gives us another view on the meaning of strategies expressed using proof terms. Figure 5 defines the *denotational semantics* of elementary strategies as a function $\mathcal{D} : \mathcal{ES} \to 2^S$, i.e. a transformation of the elementary strategy into a subset of strategies. A similar denotational function was presented in [Vit94]. Our definition gives a model for **dk** and **dc**-non-deterministic computations, while the original function modelled only **dk**-non-determinism. To illustrate the difference between **dc**, **dk**, **case**, let

$$
\begin{aligned}
s_1 \# s_2 &= s_1 \cup \{\pi \in s_2 \mid dom(\pi) \notin dom(s_1)\} \\
\mathcal{D}(\mathbf{id}) &= \{\mathcal{T}(\mathcal{F})\} \\
\mathcal{D}(\mathbf{dc}(es_1, es_2)) &= \{s_1 \# s_2, s_2 \# s_1 \mid s_i \in \mathcal{D}(es_i)\} \\
\mathcal{D}(\mathbf{dk}(es_1, es_2)) &= \{s_1 \cup s_2 \mid s_i \in \mathcal{D}(es_i)\} \\
\mathcal{D}(\mathbf{case}(es_1, es_2)) &= \{s_1 \# s_2 \mid s_i \in \mathcal{D}(es_i)\} \\
\mathcal{D}(\mathbf{f}(es_1, \ldots, es_n)) &= \{\{f(\pi_1, \ldots, \pi_n) \mid \pi_1 \in s_1, \pi_n \in s_n\} \mid s_i \in \mathcal{D}(es_i)\} = \\
&\quad \{\mathbf{f}(s_1, \ldots, s_n) \mid s_i \in \mathcal{D}(es_i)\} \\
\mathcal{D}(\mathbf{l}(es_1, \ldots, es_n)) &= \{\{l(\pi_1, \ldots, \pi_n) \mid \pi_1 \in s_1, \pi_n \in s_n\} \mid s_i \in \mathcal{D}(es_i)\} = \\
&\quad \{\mathbf{l}(s_1, \ldots, s_n) \mid s_i \in \mathcal{D}(es_i)\} \\
\mathcal{D}(es_1; es_2) &= \{\{\pi_j; \pi'_j \mid \pi_j \in s_1, \pi'_j \in u_j, u_j \in Q, j \in I\} \mid s_1 \in \mathcal{D}(es_1), \\
&\quad s_1 = \{\pi_i, i \in I\},\ Q = \{u_i \mid i \in I, u_i \in \mathcal{D}(es_2), \\
&\quad\quad (\ cod(\pi_i) \in dom(u_i)\ \vee\ cod(\pi_i) \notin dom(es_2)\ )\}\}
\end{aligned}
$$

Fig. 5. Denotational semantics of $\mathcal{ES}$

us evaluate $\mathcal{D}$ for:

$$\mathcal{D}(\mathbf{dc}(a,\ (a;a))) = \{\{a\}, \{a;a\}\} \qquad \mathcal{D}(\mathbf{case}(a,\ (a;a))) = \{\{a\}\}$$

$$\mathcal{D}(\mathbf{dk}(a,\ (a;a))) = \{\{a,\ (a;a)\}\} \qquad \mathcal{D}(\mathbf{case}((a;a),\ a)) = \{\{a;a\}\}$$

The following lemma links together the operational and denotational semantics. Its proof with more examples can be found in [Bor96].

Lemma 3.5 *(Soundness and completeness)*
For any elementary strategy $es \in \mathcal{ES}$ *and any term* $t \in \mathcal{T}(\mathcal{F})$, $res = (es\lceil t \rceil)_{\downarrow}$ *if and only if there is* $s \in \mathcal{D}(es)$, *such that* $res = s[t]$.

Now, we have two semantics, equivalent in the sense of Lemma 3.5. In general, for the purpose of strategy evaluation, it is more natural to use the operational semantics, however for strategy transformations, it is better to deal with the denotational semantics. As examples of elementary strategy transformations, the following lemma states several properties of strategy operators. The proofs or counter-examples for these equivalences or disequivalences can be found in [Bor96].

Lemma 3.6 *Let us define* $es_1 \equiv es_2$ *by* $\mathcal{D}(es_1) = \mathcal{D}(es_2)$. *Then:*

- **replacement of f by l**
  1) $\mathbf{f}(es_1, \ldots, es_n) \equiv \mathbf{l_f}(es_1, \ldots, es_n)$, *where* $[l_{\mathbf{f}}(\overline{x})] \quad f(\overline{x}) \Rightarrow f(\overline{x})$

- **removing dc**
  2) $\mathbf{dc}(es_1, es_2) \equiv \mathbf{dk}(\mathbf{case}(es_1, es_2), \mathbf{case}(es_2, es_1))$

- **distributivity of f, l on ;**
  3a) $\mathbf{f}(es_1, \ldots, es_i'; es_i'', \ldots, es_n) \not\equiv$
  $\mathbf{f}(es_1, \ldots, es_i, \ldots, es_n)$; $\mathbf{f}(es_1, \ldots, es_i'', \ldots, es_n)$
  3b) $\mathbf{l}(es_1, \ldots, es_i'; es_i'', \ldots, es_n) \not\equiv$
  $\mathbf{l}(es_1, \ldots, es_i, \ldots, es_n)$; $\mathbf{l}(es_1, \ldots, es_i'', \ldots, es_n)$

- *4a)* $\mathbf{f}(es_1, \ldots, es, \ldots, es_n) \equiv$
  $\mathbf{f}(es_1, \ldots, \mathbf{id}, \ldots, es_n)$; $\mathbf{f}(\mathbf{id}, \ldots, \mathbf{id}, es, \mathbf{id}, \ldots, \mathbf{id})$
  4b) $\mathbf{l}(es_1, \ldots, es, \ldots, es_n) \not\equiv$
  $\mathbf{l}(es_1, \ldots, \mathbf{id}, \ldots, es_n)$; $\mathbf{l}(\mathbf{id}, \ldots, \mathbf{id}, es, \mathbf{id}, \ldots, \mathbf{id})$

- **decomposition of $\mathbf{l}(es_1, \ldots, es_n)$ into $\mathbf{l}(\mathbf{id}, \ldots, \mathbf{id})$, where** $[l(\overline{x})]\ u \Rightarrow u'$
  5a) $\mathbf{l}(es_1, \ldots, es_n) \equiv \{x_i \mapsto es_i\}u$; $\mathbf{l}(\mathbf{id}, \ldots, \mathbf{id})$
  5b) $\mathbf{l}(es_1, \ldots, es_n) \not\equiv \mathbf{l}(\mathbf{id}, \ldots, \mathbf{id})$; $\{x_i \mapsto es_i\}u'$

- **distributivity of f, l on dc, dk, case**
  6a) $\mathbf{f}(es_1, \ldots, \mathbf{dk}(es_i', es_i''), \ldots, es_n) \equiv$
  $\mathbf{dk}(\mathbf{f}(es_1, \ldots, es_i, \ldots, es_n), \mathbf{f}(es_1, \ldots, es_i'', \ldots, es_n))$
  6b) $\mathbf{f}(es_1, \ldots, \mathbf{dc}(es_i', es_i''), \ldots, es_n) \equiv$
  $\mathbf{dc}(\mathbf{f}(es_1, \ldots, es_i, \ldots, es_n), \mathbf{f}(es_1, \ldots, es_i'', \ldots, es_n))$
  6c) $\mathbf{f}(es_1, \ldots, \mathbf{case}(es_i', es_i''), \ldots, es_n) \equiv$
  $\mathbf{case}(\mathbf{f}(es_1, \ldots, es_i, \ldots, es_n), \mathbf{f}(es_1, \ldots, es_i'', \ldots, es_n))$
  the same properties hold for l,

- **distributivity of ; on dk, dc, case**
  7a) es ; $\mathbf{dk}(es', es'') \equiv \mathbf{dk}(es$; es', es ; $es'')$
  7b) es ; $\mathbf{dc}(es', es'') \not\equiv \mathbf{dc}(es$; es', es ; $es'')$
  7c) es ; $\mathbf{case}(es', es'') \not\equiv \mathbf{case}(es$; es', es ; $es'')$.

4 Axiomatisation of many-sorted strategies

This section provides an axiomatisation via many-sorted rewrite theories of elementary and user-defined strategies. We first present an embedding of the user's many-sorted rewrite theory (considered as the object level) into a rewrite theory for elementary strategies in Section 4.1. Another enrichment in Section 4.2 yields the definition of rewrite theories for defined strategies, whose operational semantics is thus provided by user-defined rewrite rules. This idea of defined strategy language is compatible with the concept of the internal strategy language of [CM96], and it allows the definition of meta-strategies. Then we discuss several simplification and implementation issues in Section 4.3 and give the ELAN program of the strategy interpreter. Section 4.4 gives examples of strategies written in ELAN.

The goal of the defined strategy language is to be able to define strategies, like **map**, by a strategy rewrite rule in the following way:

$$\mathbf{map}(x) \Rightarrow \mathbf{dc}(\mathbf{nil}, x.\mathbf{map}(x))$$

The right-hand side of this definition means that whenever the strategy **map** with an argument s (i.e. $\mathbf{map}(s)$) is applied to a term t, either t is nil, or the strategy s is applied on the head of t (i.e. t should be a non-empty list) and $\mathbf{map}(s)$ is further applied on the tail of t. This strategy definition substantially differs from the traditional functional definition (cf. [MJ95] for instance) of the functor $\mathbf{map} : (a \to a) \to (list[a] \to list[a])$:

$$\begin{aligned} \mathbf{map}\ f\ nil &= nil \\ \mathbf{map}\ f\ (a.as) &= (f\ a).(\mathbf{map}\ f\ as) \end{aligned}$$

which can be reformulated in our framework using the strategy application symbol $_\lceil_\rceil$:

$$\begin{aligned} \mathbf{map}(s)\lceil nil\rceil &\Rightarrow nil \\ \mathbf{map}(s)\lceil a.as\rceil &\Rightarrow s\lceil a\rceil.\mathbf{map}(s)\lceil as\rceil \end{aligned}$$

The difference relies on the fact that a list, which the functional **map** is applied on, is an explicit argument in the second (or third) definition, while in the first one, it is implicit. One can object that the functional definition viewed as a rewrite theory is convergent, while the first one does not terminate. A natural solution of this problem inspired by ELAN's idea of labelled and unlabelled rules, is to separate the convergent and the divergent parts of a rewrite theory. Rules of the convergent part may be freely used for the normalisation of strategy expressions. Examples of normalisation rules for elementary strategies are:

$$\begin{aligned} \mathbf{id}\ ;\ es &\Rightarrow es & es\ ;\ \mathbf{id} &\Rightarrow es \\ \mathbf{dc}(es,\ es) &\Rightarrow es & \mathbf{dk}(es,\ es) &\Rightarrow es \end{aligned}$$

or can also be found in Lemma 3.6 by orienting equivalences. Normalisation rules can also express some well-known equivalences of defined strategies, e.g.

the distributivity of **map** on the concatenation operator ;

$$\mathbf{map}(s_1)\ ;\ \mathbf{map}(s_2) \Rightarrow \mathbf{map}(s_1\ ;\ s_2)$$
$$\mathbf{map}(\mathbf{id}) \Rightarrow \mathbf{id}$$

Rules of the divergent part should be applied under certain restrictions, which invokes the concept of meta-strategies, i.e. strategies which control the execution by rewriting of defined strategies.

Getting back to our first *map* definition $\mathbf{map}(x) \Rightarrow \mathbf{dc}(\mathbf{nil}, x.\mathbf{map}(x))$ it is necessary to clarify:

- *The origin and signature* of all function symbols: the overloaded symbol **nil** in this definition is a strategy term, not an object level term standing for empty list. To stress this fact, we use a bold face font for strategy symbols.
- *The type* of introduced strategy terms: if **nil** is not a list, what is its sort ?
- *The semantics* of such definitions, i.e. how do we compute an application of a strategy on a term when we have two rewrite theories, one for terms from $\mathcal{T}(\mathcal{F})$, and the second for strategies ?
- *The implementation* of such strategies. As already said, we transform strategy definitions into computational systems.

In the following sections, we answer these questions by introducing two rewrite theories: one for elementary strategies and one for defined strategies.

4.1 Rewrite theory of elementary strategies

Let us assume that the object level is a many-sorted rewrite theory:

$$\mathcal{RT} = (\Sigma, \mathcal{L}, \mathcal{R}) \qquad \text{where } \Sigma = (\mathcal{S}, \mathcal{F})$$

The rewrite theory of elementary strategies $\mathcal{RT}_{\mathcal{ES}} = (\Sigma_{\mathcal{ES}}, \mathcal{L}_{\mathcal{ES}}, \mathcal{R}_{\mathcal{ES}})$ extends the theory $\mathcal{RT}$ by adding strategy sorts, elementary strategy symbols and labelled strategy interpreter's rules in the following way:

$$\Sigma_{\mathcal{ES}} = (\mathcal{S}_{\mathcal{ES}}, \mathcal{F}_{\mathcal{ES}}) \qquad \mathcal{S}_{\mathcal{ES}} = \mathcal{S} \cup \mathcal{S}_{\mathcal{E}} \cup \mathcal{S}_{\mathcal{I}} \qquad \mathcal{F}_{\mathcal{ES}} = \mathcal{F} \cup \mathcal{F}_{\mathcal{E}} \cup \mathcal{F}_{\mathcal{I}}$$
$$\mathcal{L}_{\mathcal{ES}} = \mathcal{L} \cup \mathcal{L}_{\mathcal{E}} \cup \mathcal{L}_{\mathcal{I}} \qquad \mathcal{R}_{\mathcal{ES}} = \mathcal{R} \cup \mathcal{R}_{\mathcal{E}} \cup \mathcal{R}_{\mathcal{I}}$$

4.1.1 Sorts

$\mathcal{S}_{\mathcal{ES}}$ is the disjoint union of $\mathcal{S} \cup \mathcal{S}_{\mathcal{E}} \cup \mathcal{S}_{\mathcal{I}}$ defined as follows:
$\underline{\mathcal{S}_{\mathcal{E}}} = \{\langle s' \mapsto s'' \rangle \mid s',\ s'' \in \mathcal{S}\}$ contains new strategy sorts $\langle s' \mapsto s'' \rangle$ such that s is *a subsort* of $\langle s \mapsto s \rangle$ for any $s \in \mathcal{S}$. $\langle s' \mapsto s'' \rangle$ is the sort of all elementary strategy terms transforming terms of sort s' into terms of sort s''. If all rules are sort preserving, we do not need to include sorts $\langle s' \mapsto s'' \rangle$ for $s' \neq s''$. A natural question is whether it is useful to generalise this rewrite theory by introducing sort-changing strategies. This can be motivated by an example: if the argument x of **map** is instantiated by a sort-changing strategy of sort $\langle s' \mapsto s'' \rangle$, $\mathbf{map}(x)$ is a strategy which transforms $list[s']$ into $list[s'']$.
$\underline{\mathcal{S}_{\mathcal{I}}}$: consists of sorts used in the interpreter (like $\mathbf{List}[s]$, booleans, etc.).

4.1.2 Functions

$\mathcal{F}_{\mathcal{ES}}$ is the disjoint union of $\mathcal{F} \cup \mathcal{F}_{\mathcal{E}} \cup \mathcal{F}_{\mathcal{I}}$ defined as follows:

$\mathcal{F}_{\mathcal{E}}$: To explain the definition of $\mathcal{F}_{\mathcal{E}}$ given below, let us consider first the following problem: if we allow sort-changing strategies, the three f's in the **Congruence** rule are different symbols. Let us look at a *cons*-instantiation of this rule:

$$\frac{es_h : h \Rightarrow h',\ es_t : t \Rightarrow t'}{\mathbf{cons}(es_h, es_t) : cons(h,t) \Rightarrow cons(h',t')}$$

If es_h is a strategy transforming a term of sort s' into a term of sort s'' (i.e. $es_h : \langle s' \mapsto s'' \rangle$) and $es_t : \langle list[s'] \mapsto list[s''] \rangle$, then *cons* on the left of $\Rightarrow$ has rank $(s'\ list[s']) \mapsto list[s']$, while *cons* on the right of $\Rightarrow$ has rank $(s''\ list[s'']) \mapsto list[s'']$. The **cons** constructor of proof terms has then rank $(\langle s' \mapsto s'' \rangle\ \langle list[s'] \mapsto list[s''] \rangle) \mapsto \langle list[s'] \mapsto list[s''] \rangle$. The following definition generalises this example.

The set of elementary strategy symbols $\mathcal{F}_{\mathcal{E}}$ consists of the following symbols:

$$\begin{aligned}
\mathbf{f} \ :\ & (\langle s'_1 \mapsto s''_1 \rangle \dots \langle s'_n \mapsto s''_n \rangle) \mapsto \langle s' \mapsto s'' \rangle \\
& \quad \textit{there is a pair of } f\textit{'s} \ : \\
& \quad f : (s'_1 \dots s'_n) \mapsto s' \in \Sigma, \text{ and } f : (s''_1 \dots s''_n) \mapsto s'' \in \Sigma \\
\mathbf{l} \ :\ & (s'_1 \dots s'_n) \mapsto \langle s' \mapsto s'' \rangle \\
& \quad \text{if } [l(x_1 : s_1, \dots, x_n : s_n)]\ u : s' \Rightarrow u' : s'' \in \mathcal{R} \text{ and} \\
& \quad s'_i = \langle s_i \mapsto s_i \rangle \text{ if } x_i \in Var(u) \cap Var(u'), \quad \textit{otherwise } s'_i = s_i \\
\mathbf{dk},\ \mathbf{dc},\ \mathbf{case} \ :\ & (\langle s' \mapsto s'' \rangle\ \langle s' \mapsto s'' \rangle) \mapsto \langle s' \mapsto s'' \rangle \\
; \ :\ & (\langle s' \mapsto s'' \rangle\ \langle s'' \mapsto s''' \rangle) \mapsto \langle s' \mapsto s''' \rangle \\
\mathbf{id} \ :\ & \langle s \mapsto s \rangle
\end{aligned}$$

Example 4.1 With the traditional definition of the sort $list[X]$ with two constructors (nil, $_._$) and a rewrite rule: $[lab(x : X, u : X)]\quad x + 0 \Rightarrow x + u$, the previous definition generates the following strategy symbols:

$$\begin{aligned}
\mathbf{nil} \ :\ & \langle list[X] \rangle \\
_._ \ :\ & (\langle X \rangle\ \langle list[X] \rangle) \mapsto \langle list[X] \rangle \\
\mathbf{lab}(_,_) \ :\ & (\langle X \rangle\ X) \mapsto \langle X \rangle
\end{aligned}$$

where $\langle s \rangle$ stands for $\langle s \mapsto s \rangle$. This example becomes more interesting when using simultaneously two list sorts, i.e. $list[X]$ and $list[Y]$, because we have to add not only similar strategy symbols for $list[Y]$, but also

$$\begin{aligned}
\mathbf{nil} \ :\ & \langle list[X] \mapsto list[Y] \rangle \\
_._ : \ & (\langle X \mapsto Y \rangle\ \langle list[X] \mapsto list[Y] \rangle) \mapsto \langle list[X] \mapsto list[Y] \rangle
\end{aligned}$$

and vice-versa (for $\langle Y \mapsto X \rangle$ and $\langle list[Y] \mapsto list[X] \rangle$). For a better orientation in this overloaded jungle, we add unique decorations to all function symbols of $\mathcal{F}$ (for example nil^X, nil^Y), and we join a pair of these decorations

with each strategy symbol from $\mathcal{F}_{\mathcal{E}}$ to indicate their 'parents' (for example $\mathbf{nil}^{X,X}$, $\mathbf{nil}^{Y,Y}$, $\mathbf{nil}^{X,Y}$, $\mathbf{nil}^{Y,X}$).

$\underline{\mathcal{F}_{\mathcal{I}}}$: contains different function symbols used in the interpreter. In particular, the symbol for application of a strategy to a term has the following rank:

$$_\lceil_\rceil \ : \ (\langle s' \mapsto s'' \rangle \ s') \mapsto \mathbf{List}[s'']$$

4.1.3 Rewrite rules

$\mathcal{R}_{\mathcal{ES}}$ is the disjoint union of $\mathcal{R} \cup \mathcal{R}_{\mathcal{E}} \cup \mathcal{R}_{\mathcal{I}}$ defined as follows:

$\underline{\mathcal{R}_{\mathcal{E}}}$: consists of two sets of strategy rewrite rules FSYM and LAB defining the semantics of the symbol $_\lceil_\rceil$.

- FSYM rules give the interpretation of applying a strategy $\mathbf{f}(es_1, \ldots, es_n)$ on a term. For any symbol $\mathbf{f}$ such that there is a pair of its 'parents' $f : (s'_1 \ldots s'_n) \mapsto s' \in \Sigma$ and $f : (s''_1 \ldots s''_n) \mapsto s'' \in \Sigma$, the following rewrite rule over sort s'' is generated:

$$[\mathsf{FSYM}_{\mathbf{List}[s'']}]\mathbf{f}(S_1, \ldots, S_n)\lceil f(x_1, \ldots, x_n)\rceil \Rightarrow join_f(y_1, \ldots, y_n)$$
$$\text{where } y_i := (S_i\lceil x_i\rceil)_{\downarrow}, \ i = 1..n$$

  where variables have types: $S_i : \langle s'_i \mapsto s''_i \rangle$, $x_i : s'_i$ and $y_i : \mathbf{List}[s''_i]$. The label FSYM of this rule refers to the set of all rules generated by this schema, and its index $\mathbf{List}[s'']$ classifies the set of FSYM rules by the common type of their left and right-hand sides. This label will be later useful for the definition of meta-level strategies of the interpreter.

  Example 4.2 The previously mentioned decorations help us to understand that there are four FSYM rules for the strategy symbol **nil**, obtained from the rule:

$$[\mathsf{FSYM}_{\mathbf{List}[Y]}] \quad \mathbf{nil}^{X,Y}\lceil nil^X\rceil \Rightarrow nil^Y$$

  by varying X and Y. The situation with the symbol $_\ldots_$ is similar. These decorations also show that the f's in FSYM schema are in general different [3].

  A brute force approach would generate $\mathcal{O}(|\mathcal{S}|^3 + |\mathcal{F}|^2)$ strategy symbols and FSYM rules. To prevent this explosion, the user should have the possibility to give strategy declarations for the strategies (s)he wants to use, and only then, strategy symbols and rules over these 'declared strategies' are automatically generated.

- LAB rules give the interpretation of applying a strategy $\mathbf{l}(es_1, \ldots, es_n)$ on a term. For any rewrite rule from $\mathcal{R}$ of the form:

$$[l(x_1 : s_1, \ldots, x_n : s_n)] \quad u : s' \Rightarrow u' : s''$$

  where $x_i \in Var(u) \cap Var(u')$ for $1 \leq i \leq m$, the following labelled rewrite rule over the sort $\mathbf{List}[s'']$ is embedded:

$$[\mathsf{LAB}_{\mathbf{List}[s'']}] \ \mathbf{l}(S_1, \ldots, S_m, x_{m+1}, \ldots, x_n)\lceil u(x_1, \ldots, x_m)\rceil \Rightarrow$$
$$join_{u'}(y_1, \ldots, y_m, x_{m+1}, \ldots, x_n) \quad \text{where } y_i := (S_i\lceil x_i\rceil)_{\downarrow}, \ i = 1..m$$

[3] Take off your overloaded glasses!

Again the label LAB refers to the set of all rules generated by this schema, and its index $\mathbf{List}[s'']$ classifies the set of LAB rules by the common type of their results. This label will also be useful later on for the definition of meta-level strategies.

Example 4.3 Continuing our example, one rewrite rule is generated for the rule *lab* (supposing that the signature of the symbol + is $(X\ X) \mapsto X$):

$$\mathbf{lab}(_,_) \quad : (\langle X\rangle\ X) \mapsto \langle X\rangle$$

$$[\mathsf{LAB}_{\mathbf{List}[X]}]\ \mathbf{lab}(S_1, x_2)\lceil x_1 + 0\rceil \Rightarrow join_{y_1+x_2}(y_1, x_2) \quad \text{where } y_1 := (S_1\lceil x_1\rceil)_{\downarrow}$$

$\underline{\mathcal{R}_{\mathcal{I}}}$: consists of the interpreter rules described in Figure 3 defining the semantics of **dk**, **case**, **id**, etc.
$\mathcal{L}_{\mathcal{ES}}$ is the disjoint union of $\mathcal{L} \cup \mathcal{L}_{\mathcal{E}} \cup \mathcal{L}_{\mathcal{I}}$ defined as follows:
$\underline{\mathcal{L}_{\mathcal{E}}}$: contains labels of newly added rules, i.e. $\mathsf{FSYM}_{\mathbf{List}[s'']}$ and $\mathsf{LAB}_{\mathbf{List}[s'']}$.
$\underline{\mathcal{L}_{\mathcal{I}}}$: contains the labels of the interpreter rules of $\mathcal{R}_{\mathcal{I}}$ described in Figure 3.

4.2 Rewrite theory for defined strategies

Finally, we introduce the notion of *defined strategies*, whose syntax is built on the top of strategy symbols $\mathcal{F}_{\mathcal{ES}}$. Let us suppose that there is a set of *defined strategy symbols* $\mathcal{F}_{\mathcal{D}} = \cup_n \mathcal{F}_{\mathcal{D}_n}$ with signatures:

$$\mathbf{d} : (s_1 \ldots s_n) \mapsto \langle s' \mapsto s''\rangle \qquad \text{if } d \in \mathcal{F}_{\mathcal{D}_n}$$

where $s_i \in \mathcal{S}_{\mathcal{ES}}$ and $s', s'' \in \mathcal{S}$. $X_{\mathcal{D}}$ is a set of *strategy variables* ($\mathcal{X}_{\mathcal{DS}} = \mathcal{X}_{\mathcal{D}} \cup \mathcal{X}$) which range over strategy sorts $\langle s'_i \mapsto s''_i\rangle \in \mathcal{S}_{\mathcal{E}}$, while object variables range over sorts $s_i \in \mathcal{S}$.

A strategy definition is a finite set of rewrite rules of the form $sh \Rightarrow sb$ where the strategy head sh has a top symbol in $\mathcal{F}_{\mathcal{D}}$, i.e. $sh \in \{d(es_1, \ldots, es_n) \mid d \in \mathcal{F}_{\mathcal{D}},\ es_i \in \mathcal{T}(\mathcal{F}_{\mathcal{ES}} \cup \mathcal{L} \cup \mathcal{X}_{\mathcal{DS}})\}$, and the strategy body sb is a term built from defined strategy symbols $\mathcal{F}_{\mathcal{DS}}$, rule labels $\mathcal{L}$ and variables $\mathcal{X}_{\mathcal{DS}}$, such that $Var(sb) \subseteq Var(sh)$.
$\underline{\mathcal{R}_{\mathcal{D}}}$: consists of defined strategy rules:

$$[\mathsf{DSTR}_{\mathbf{List}[s'']}]\quad \mathbf{d}(S_1, \ldots, S_n)\lceil self\rceil \Rightarrow ds\lceil self\rceil \quad \text{if } S_i = ds_i,\ i = 1..n$$

for any strategy definition rule $\mathbf{d}(ds_1, \ldots, ds_n) \Rightarrow ds$ of the symbol $\mathbf{d} : (s_1\ \ldots\ s_n) \mapsto \langle s' \mapsto s''\rangle$. $self : s'$ is a new variable in $\mathcal{X}$, and S_i has the type s_i. Labels $\mathsf{DSTR}_{\mathbf{List}[s'']}$ belong to $\mathcal{L}_{\mathcal{D}}$.

Example 4.4 Continuing our example, for the definition of **map**, we add

$$[\mathsf{DSTR}_{\mathbf{List}[list[X]]}]\quad \mathbf{map}(S)\lceil self\rceil \Rightarrow \quad \mathbf{dc}(\mathbf{nil}, S.\mathbf{map}(S))\lceil self\rceil$$

Summary: The rewrite theory of defined strategies is defined by $\mathcal{RT}_{\mathcal{DS}} = (\Sigma_{\mathcal{DS}}, \mathcal{L}_{\mathcal{DS}}, \mathcal{R}_{\mathcal{DS}})$:

$$\Sigma_{\mathcal{DS}} = (\mathcal{S}_{\mathcal{ES}}, \mathcal{F}_{\mathcal{DS}}) \qquad \mathcal{F}_{\mathcal{DS}} = \mathcal{F}_{\mathcal{ES}} \cup \mathcal{F}_{\mathcal{D}}$$

$$\mathcal{L}_{\mathcal{DS}} = \mathcal{L}_{\mathcal{ES}} \cup \mathcal{L}_{\mathcal{D}} \qquad \mathcal{R}_{\mathcal{DS}} = \mathcal{R}_{\mathcal{ES}} \cup \mathcal{R}_{\mathcal{D}}$$

Now, we define the operational semantics of defined strategies as a function:

$$\mathcal{O} \quad : \quad \mathcal{DS} \times \mathcal{T}(\mathcal{F}) \to 2^{2^{\mathcal{T}(\mathcal{F})}}$$

$$\mathcal{O}(ds, t) = \{res \subseteq \mathcal{T}(\mathcal{F}) \mid res = (ds\lceil t\rceil)_{\downarrow \mathcal{R}_{\mathcal{DS}}}\} \text{ where } ds \in \mathcal{DS} \text{ and } t \in \mathcal{T}(\mathcal{F}).$$

Example 4.5 The following example shows several basic strategy definitions.

variables	$x : \langle s \rangle \quad xs : \langle list[s] \rangle \quad t : \langle u \mapsto v \rangle$		
while	$: (\langle s \rangle) \mapsto \langle s \rangle$	**while**(x)	$\Rightarrow$ **dk**$(x;$ **while**$(x),$ **id**$)$
dcwhile	$: (\langle s \rangle) \mapsto \langle s \rangle$	**dcwhile**(x)	$\Rightarrow$ **dc**$(x;$ **dcwhile**$(x),$ **id**$)$
repeat	$: (\langle s \rangle) \mapsto \langle s \rangle$	**repeat**(x)	$\Rightarrow$ **case**$(x;$ **repeat**$(x),$ **id**$)$
map1	$: (\langle s \rangle) \mapsto \langle list[s] \rangle$	**map1**(x)	$\Rightarrow$ **dc**$($**nil**$, x.$**map1**$(x))$
map2	$: (\langle list[s] \rangle) \mapsto \langle list[s] \rangle$	**map2**(nil)	$\Rightarrow$ **nil**
		map2$(x.xs)$	$\Rightarrow x.$**map2**(xs)
apply	$: (\langle u \mapsto v \rangle) \mapsto \langle list[u] \mapsto list[v] \rangle$	**apply**(t)	$\Rightarrow$ **dc**$($**nil**$, t.$**apply**$(t))$

while strategy differs from **repeat** in a such way that it returns all intermediate forms during the normalisation of a term by the application of a strategy x, while **repeat** returns only the last one (i.e. normal form). The strategy **map1** applies a fixed strategy x on all elements of a list and produces a new list of transformed elements. The strategy **map2** is driven by a list of strategies which are respectively applied to elements of a list of same length. A nice example showing how deep we are in the overloaded jungle, is the comparison of **map1** and **apply**. The two rules are syntactically equivalent, up to renaming, however semantically different (*nil*'s in both are not the same) and differently typed (see their ranks).

Example 4.6 Having a rewrite rule: $[lab(x : X)] \quad x + 0 \Rightarrow x$ and all strategies defined above, all results of $(ds\lceil t\rceil)_{\downarrow \mathcal{DS}}$ are listed bellow:

$ds\lceil t\rceil$	$\mathcal{O}(ds, t)$
while(lab(id))$\lceil(a+0)+0\rceil$	$\{(a+0)+0, a+0, a\}$
dcwhile(lab(id))$\lceil(a+0)+0\rceil$	$\{(a+0)+0\},\ \{a+0\},\ \{a\}$
repeat(lab(id))$\lceil(a+0)+0\rceil$	$\{a\}$
map2(while(lab(id)).dcwhile(lab(id)).*nil*$)\lceil a+0.b+0.nil\rceil$	$\{a+0.b.nil, a.b.nil\},\ \{a+0.b+0.nil, a.b+0.nil\}$

4.3 Implementation in ELAN

The interpreter described in Figures 3 and 4 has been first implemented in ELAN as a non-deterministic rewrite theory. It computes several results corresponding to different **dc**-choices made during the computation. Each of them represents a set of different solutions corresponding to **dk**-choices. This

implementation approach is adequate for modelling semantics, however not realistic for the purpose of a real controlling language. For this reason, we improved it using ELAN strategies. We implement **dc**-non-determinism such that it takes always the first found solution; **dk**-non-determinism is modelled by backtracking of a low-level ELAN strategy. The advantage of this step (from a pure rewrite theory to a computational system) is that if we are able to guarantee, for example using ELAN strategies, certain conditions on the application of rewrite rules, we can remove some guarding conditions from the rules. For example, it is clear that the two rules of the interpreter labelled by FSYM_1, FSYM_2 are mutually exclusive, thanks to their conditions and left-hand sides. Thus, we can remove the condition $f \neq g$ or $m \neq n$, if we assume that these rules are applied with an ELAN strategy dont-care-choose(FSYM_1 FSYM_2) trying them in this order. In such a way, we obtain a more efficient interpreter as a computational system, that however preserves the semantics of the original rewrite theory. The rules FSYM, LAB, DSTR are implemented as follows:

$$_\lceil_\rceil \quad : (\langle s' \mapsto s'' \rangle\ s') \mapsto s''$$

$$[\mathsf{FSYM}_{s''}]\ \mathbf{f}(S_1,\ldots,S_n)\lceil f(x_1,\ldots,x_n)\rceil \Rightarrow f(y_1,\ldots,y_n) \quad \text{where } y_i := S_i\lceil x_i\rceil,\ i = 1..n$$

$$[\mathsf{LAB}_{s''}]\ \mathbf{l}(S_1,\ldots,S_m,x_{m+1},\ldots,x_n)\lceil u(x_1,\ldots,x_m)\rceil \Rightarrow u'(y_1,\ldots,y_m,x_{m+1},\ldots,x_n) \quad \text{where } y_i := S_i\lceil x_i\rceil,\ i = 1..m$$

$$[\mathsf{DSTR}_{s''}]\ \mathbf{d}(S_1,\ldots,S_n)\lceil self\rceil \Rightarrow ds\lceil self\rceil \quad \text{if } S_i = ds_i,\ i = 1..n$$

where variables have types: $S_i : \langle s'_i \mapsto s''_i \rangle$, $x_i : s'_i$, $y_i : s''_i$ and $self : s'$. The rules of the interpreter (Figure 6) are defined in an ELAN module parameterised by three sorts s, s', s''. This set of rules *has to be controlled* by the ELAN strategy $eval_{s''}$ described in Figure 7 and written in the ELAN strategy language. Informally, `dont-know/care-choose` in Figure 7 stands for **dk**, resp. **dc**, || separates alternatives and the strategy `repeat-endrepeat` stands for the ELAN's built-in strategy corresponding to our definition of **repeat**. The non-deterministic strategy $eval_{s''}$ is used whenever an application of a strategy $S : \langle s' \mapsto s'' \rangle$ on a term $t : s'$ is evaluated, i.e. whenever $(S\lceil t\rceil)_{\downarrow}$ is computed. This gives also the operational meaning for ' where ' expressions used in FSYM and LAB rules, i.e. an expression where $y_i := S_i\lceil x_i\rceil$ in the FSYM rule means that y_i is instantiated to all reductions of $S_i\lceil x_i\rceil$ by ELAN strategy $eval_{s''_i}$.

The rewrite theories built up to now have been formulated as ELAN's theories and strategies. However, ELAN's theories include rewrite rules with ' where ' expressions and thus are slightly different from pure conditional rewrite theories described in [Mes92,MOM93]. We propose in [Bor96] a translation of ELAN rules into pure rewriting logic and express the strategy *eval* using defined strategies.

module $strcommons[s, s', s'']$

rules for s''

declare $x : s;\ y : s';\ z : s'';\quad S_0 : \langle s \mapsto s' \rangle;\quad S_1,\ S_2 : \langle s' \mapsto s'' \rangle$

bodies

$[\mathsf{DK}_{s''}]$ $\quad \mathbf{dk}(S_1, S_2)\lceil y \rceil \Rightarrow z$ where $z := S_1\lceil y \rceil$

$[\mathsf{DK}_{s''}]$ $\quad \mathbf{dk}(S_1, S_2)\lceil y \rceil \Rightarrow z$ where $z := S_2\lceil y \rceil$

$[\mathsf{ID}_{s''}]$ $\quad \mathbf{id}\lceil y \rceil \Rightarrow \quad y$

$[\mathsf{DC1}_{s''}]$ $\quad \mathbf{dc}(S_1, S_2)\lceil y \rceil \Rightarrow z$ where $z := S_1\lceil y \rceil$

$[\mathsf{DC2}_{s''}]$ $\quad \mathbf{dc}(S_1, S_2)\lceil y \rceil \Rightarrow z$ where $z := S_2\lceil y \rceil$

$[\mathsf{CONC}_{s''}]$ $\quad S_0; S_1\lceil x \rceil \Rightarrow \quad z$ where $z := S_1\lceil y \rceil$ where $y := S_0\lceil x \rceil$

$[\mathsf{FIN}_{s''}]$ $\quad z \Rightarrow \quad z$ if $z \neq S_1\lceil y \rceil$

end of rules

Fig. 6. Interpreter in ELAN

strategy $eval_{s''}$ *for* s''

repeat

dont care choose(

dont know choose($\mathsf{FSYM}_{s''}$) || *dont know choose*($\mathsf{LAB}_{s''}$) ||

dont know choose($\mathsf{DSTR}_{s''}$) || *dont care choose*($\mathsf{ID}_{s''}$) ||

dont know choose($\mathsf{DK}_{s''}$) || *dont know choose*($\mathsf{CONC}_{s''}$) ||

dont know choose($\mathsf{DC1}_{s''}$) || *dont know choose*($\mathsf{DC2}_{s''}$))

endrepeat

dont care choose($FIN_{s''}$)

end of strategy

Fig. 7. The ELAN strategy of the interpreter

4.4 ELAN's description of some strategies

The following example shows a basic strategy module of ELAN's library.

```
module basic[X,Y]
import global // strategy declarations
  strat[X,X] strat[list[X],list[X]]
  strat[X,Y] strat[list[X],list[Y]];
str global    // strategy profiles
  map1(_) :(<X>) <list[X]>;
  map2(_) :(<list[X]>) <list[X]>;
  apply(_):(<X->Y>) <list[X]->list[Y]>;
  while(_):(<X>) <X>;
  dcwhile(_):(<X>) <X>;
  repeat(_):(<X>) <X>;
  try _    : (<X>) <X>;
endstr
strategy for <list[X]->list[Y]>
declare t : <X -> Y>;
body
  apply(t) => dc(nil, t.apply(t))
end of strategy
```

```
strategies for <list[X]>
declare x : <X>;    xs : <list[X]>;
bodies
  map1(x) => dc(nil, x.map1(x))
  map2(nil) => nil
  map2(x.xs) => x.map2(xs)
end of strategies
strategies for <X>
declare x : <X>;
bodies
  repeat(x) =>case(x;repeat(x), id)
  while(x)  =>dk(x;while(x), id)
  dcwhile(x)=>dc(x;dcwhile(x), id)
  try x     =>case(x,id)
end of strategies
```

More sophisticated examples with a complete description of the strategy language can be found in [Bor96].

5 Describing a mechanical theorem prover in ELAN

We sketch here the implementation of an elementary mechanical theorem prover for first-order predicate calculus (FOPC). Beyond the advantage of giving more examples of strategy definitions, handling this example also provides a basis for comparisons with other logical frameworks and mechanical theorem provers, such as for instance LCF, NuPrl, HOL, ELF and 2OBJ. We do not deeply explain all details of this system, because this would be out of this paper scope, but the interested reader may look at [GSHH94] and [Bor96].

We encode a Gentzen-style sequent calculus and mimics a *proof calculator* operating over a domain of sequents. The system transforms lists of sequents of the form $H \vdash Y \,.\, G$, where the hypotheses H are a list of sentences ($list[Sent]$), uniquely labelled with $\underline{1}, \ldots, \underline{N}, \ldots$, the conclusion Y is a sentence ($Sent$), and $G : list[Goal]$ represents the rest of the sequents. The transformation is done by rules such as:

$$[\vdash exist(t)] \quad H \vdash (\exists v)Y \,.\, G \Rightarrow H \vdash \{v \mapsto t\}Y \,.\, G$$

encoding for instance the $\exists$-introduction inference rule. The parameter $t : Term$ of this rule is the term by which the bound variable $v : Var$ is replaced. Labels of these inference rules are elementary strategy operators, for instance $\vdash exist : (Term)\ \langle Goals \rangle$. This immediately provides several basic actions over sequents. Sequential compositions of already pre-cooked proofs from basic inference rules can be designed using the elementary strategy symbol ;. More sophisticated proof strategies can be constructed which correspond to proof transformations in FOPC logic. A typical example is a *cut*-elimination rule encoded as:

$$[cut(Y)] \quad H \vdash X \,.\, G \Rightarrow H \vdash Y \,.\, H; (unique_label : Y) \vdash X \,.\, G$$

or an induction application strategy $ind(v : sort) : \langle Goals \rangle$ parameterised by an induction variable with its sort, which works on the principle of cover set

induction. We can also combine several subproofs with strategy operators defined in section 4.4. But proving even small and simple theorems by searching and applying appropriate inference rules is rather cumbersome. A smarter way is to add a guide function (strategy) which helps choosing a suitable inference rule, or a strategy. The following example shows two cases in the definition of this guide strategy $concl_guide : (Goals)\ \langle Goals\rangle$, which selects a strategy applicable on the conclusion part of a sequent:

$$concl_guide(H \vdash X \to Y\ .\ G) \Rightarrow \ \vdash impl$$

$$concl_guide(H \vdash (\exists v)X\ .\ G) \Rightarrow \ \vdash exist(\text{prompt("give a substitution term:")})$$

In the second case, the proof calculator asks the user for a substitution term to complete the construction of a guided strategy. Now, we can define a guiding strategy $follow_concl_guide : \langle Goals\rangle$ by the rule:

$$follow_concl_guide \Rightarrow concl_guide(self)$$

or in a more readable *term-dependent* format:

$$follow_concl_guide\lceil G\rceil \Rightarrow concl_guide(G)\lceil G\rceil.$$

The application of $follow_concl_guide\lceil(\underline{1} : p) \vdash q \to p\rceil$ searches a candidate:

$$concl_guide((\underline{1} : p) \vdash q \to p)\lceil(\underline{1} : p) \vdash q \to p\rceil \Rightarrow \ \vdash impl\lceil(\underline{1} : p) \vdash q \to p\rceil$$

and then applies this advice $\vdash impl$ on the goal:

$$\vdash impl\lceil(\underline{1} : p) \vdash q \to p\rceil \Rightarrow (\underline{1} : p)\ ;\ (\underline{2} : q) \vdash p).$$

Our approach is extensible to definitions such as:

$$last_hyp\ :\ ((int)\ \langle Goal\rangle)\ \langle Goals\rangle$$

where a strategy *last_hyp* is parameterised by another parameterised strategy $ss : (int)\ \langle Goal\rangle$. A meaningful example is the following definition:

$$last_hyp\ ss \Rightarrow ss(max_hypothesis_label(self))$$

which might be more readable in the following form:

$$last_hyp\ ss\lceil H \vdash X\rceil \Rightarrow ss(max_label(H))\lceil H \vdash X\rceil.$$

It defines a strategy operator that completes its argument strategy by the maximal label among hypotheses.

These few examples illustrate how easily our strategy framework is applicable to mechanical theorem proving. Almost all definitions given above are similar to those given in the distributed version of 2OBJ, but one of the main differences is that our strategy application operator $_\lceil_\rceil$ has a clear defined semantics in rewriting logic, while 2OBJ uses a notion of an *expansion operator*, which has a 'side-effect' semantics [GSHH94] and is coded in LISP. Due to this, in 2OBJ it is difficult to define a tactical (a strategy) for repeating a given tactic n times. In our syntax, we write the following strategy definition:

$$\mathbf{repeat}(n, s) \Rightarrow \text{ if } n = 0 \text{ then } \mathbf{id} \text{ else } s; \mathbf{repeat}(n-1, s) \text{ fi}$$

that can be reformulated into a *goal dependent* version:

$$\mathbf{repeat}(n, s)\lceil G\rceil \Rightarrow \text{ if } n = 0 \text{ then } \mathbf{id} \text{ else } s; \mathbf{repeat}(n-1, s)\lceil G\rceil \text{ fi}$$

Another advantage of our framework is to provide typed strategies, which ensures a safer and more expressive strategy language.

6 Conclusion

Our approach for embedding strategies (as a control on rewriting) in rewriting logic can be summarised as follows: first, we have described strategies as first-order terms, we have explained their operational semantics in rewriting logic and we have shown a possible way of typing them. Finally, we have illustrated their implementation into the logical framework ELAN[4].

A prototype of the proposed strategy language, which gives us feedback for further development of the strategy theory, is incorporated directly into the system ELAN. There is a part of this implementation, which is dependent of the user's definition of strategies, and which is written in C++. This part mostly concerns the generation of elementary strategy symbols and the generation of rewrite rules (FSYM, LAB, DSTR). The independent part of the interpreter is written in ELAN itself and mostly concerns rules like DK, DC, ID, etc. Thanks to the efficient ELAN compiler [Vit96][5], the idea of transforming user's defined strategies into lower level ELAN strategies shows to be realistic. Moreover, using this bootstrapping technique, we always keep an interpreted and a compiled version of the strategy language which are coherent (w.r.t. ELAN compiler). This property seems very important in this state of the language development and prototyping.

Acknowledgements

We sincerely thank Marian Vittek and Pierre-Etienne Moreau for all our interactions on ELAN from the design to the very implementation, and José Meseguer for many valuable discussions on strategies.

References

[Bor96] P. Borovanský. Strategies for computational systems. Technical report, CRIN & INRIA-Lorraine, France, 1996.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier.

[4] The system ELAN is now available at the address
`http://www.loria.fr/equipe/protheo/PROJECTS/ELAN/ELAN.html`

[5] In many benchmarks better than compiled Caml or Sml.

[CM96] M. G. Clavel and J. Meseguer. Axiomatizing Reflective Logics and Languages. In G. Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.

[GSHH92] J. Goguen, A. Stevens, K. Hobley, and H. Hilberdink. 2OBJ, a metalogical framework based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992.

[GSHH94] J. Goguen, A. Stevens, K. Hobley, and H. Hilberdink. Mechanised Theorem Proving with 2OBJ: A Tutorial Introduction. `ftp prg.oxford.ac.uk`, 1994.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[KKV95a] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, pages 131–158. The MIT press, 1995.

[KKV95b] C. Kirchner, H. Kirchner, and M. Vittek. ELAN *V 1.17 User Manual.* INRIA Lorraine & CRIN, Nancy (France), first edition, November 1995.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MJ95] E. Meijer and J. Johan. Merging Monads and Folds for Functional Programming. In E. Meijer and J. Johan, editors, *Proceedings of Advanced Fonctional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 228–266. Springer-Verlag, 1995.

[MOM93] N. Martì-Oliet and J. Meseguer. Rewriting logic as a logical and semantical framework. Technical report, SRI International, May 1993.

[Vit94] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, October 1994.

[Vit96] M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), July 1996. Springer-Verlag.

Electronic Notes in Theoretical Computer Science 4 (1996)

Rewriting Logic as a Logical and Semantic Framework ⋆

Narciso Martí-Oliet [a,b,1] and José Meseguer [b,2]

[a] *Departamento de Informática y Automática*
Escuela Superior de Informática
Universidad Complutense de Madrid, Spain
narciso@eucmos.sim.ucm.es

[b] *Computer Science Laboratory*
SRI International
Menlo Park CA 94025, USA
meseguer@csl.sri.com

Abstract

Rewriting logic [40] is proposed as a logical framework in which other logics can be represented, and as a semantic framework for the specification of languages and systems. Using concepts from the theory of general logics [39], representations of an object logic $\mathcal{L}$ in a framework logic $\mathcal{F}$ are understood as mappings $\mathcal{L} \to \mathcal{F}$ that translate one logic into the other in a conservative way. The ease with which such maps can be defined is discussed in detail for the cases of linear logic, logics with quantifiers, and any sequent calculus presentation of a logic for a very general notion of "sequent." Using the fact that rewriting logic is reflective, it is often possible to reify inside rewriting logic itself a representation map $\mathcal{L} \to \mathit{RWLogic}$ for the finitely presentable theories of $\mathcal{L}$. Such a reification takes the form of a map between the abstract data types representing the finitary theories of $\mathcal{L}$ and of *RWLogic*.

Regarding the different but related use of rewriting logic as a semantic framework, the straightforward way in which very diverse models of concurrency can be expressed and unified within rewriting logic is illustrated with CCS. In addition, the way in which constraint solving fits within the rewriting logic framework is briefly explained.

⋆ This paper is a short version of [36], where the reader can find more examples and details not discussed here.

[1] Supported by Office of Naval Research Contract N00014-95-C-0225 and by CICYT, TIC 95-0433-C03-01.

[2] Supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, National Science Foundation Grant CCR-9224005, and by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

© *1996 Elsevier Science B. V.*

1 Introduction

The relationships between logic and computation, and the mutual interactions between both fields, are becoming stronger and more pervasive than they have ever been. In fact, our way of thinking about both logic and computation is being altered quite strongly. For example, there is such an increasingly strong connection—in some cases to the point of complete identification—between computation and deduction, and such impressive progress in compilation techniques and computing power, that the frontiers between logical systems, theorem-provers, and declarative programming languages are shifting and becoming more and more tenuous, with each area influencing and being influenced by the others.

Similarly, in the specification of languages and systems there is an increasing shift from mathematically precise but somewhat restricted formalisms towards specifications that are not only mathematical, but actually logical in nature, as exemplified, for example, by specification formalisms such as algebraic specifications and structural operational semantics. In this way, languages and systems that in principle may not seem to bear any resemblance to logical systems and may be completely "conventional" in nature, end up being conceptualized primarily as *formal* systems.

However, any important development brings with it new challenges and questions. Two such questions, that we wish to address in this paper are:

- *How can the proliferation of logics be handled?*
- *Can flexible logics allowing the specification and prototyping of a wide variety of languages and systems with naturalness and ease be found?*

Much fruitful research has already been done with the aim of providing adequate answers to these questions. Our aim here is to contribute in some measure to their ongoing discussion by suggesting that rewriting logic [40] seems to have particularly good properties recommending its use as both a *logical framework* in which many other logics can be represented, and as a general *semantic framework* in which many languages and systems can be naturally specified and prototyped.

In our view, the main need in handling the proliferation of logics is primarily conceptual. What is most needed is a *metatheory* of logics helping us to better understand and explore the boundaries of the "space" of all logics, present and future, and to relate in precise and general ways many of the logics that we know or wish to develop.

Following ideas that go back to the original work of Goguen and Burstall on *institutions* [19], we find very useful understanding the space of all logics as a *category*, with appropriate translations between logics as the arrows or morphisms between them. The theory of general logics [39] that we present in summary form in Section 2 expands the primarily model-theoretic viewpoint provided by institutions to give an adequate treatment of proof-theoretic aspects such as entailment and proof structures, and suggests not just one space or category of logics, but several, depending on the proof-theoretic or model-

theoretic aspects that we wish to focus on.

In our view, the quest for a *logical framework*, understood as a logic in which many other logics can be represented, is important but is not the primary issue. Viewed from the perspective of a general space of logics, such a quest can in principle—although perhaps not in all approaches—be understood as the search within such a space for a logic $\mathcal{F}$ such that many other logics $\mathcal{L}$ can be represented in $\mathcal{F}$ by means of mappings $\mathcal{L} \longrightarrow \mathcal{F}$ that have particularly nice properties such as being conservative translations.

Considered in this way, and assuming a very general axiomatic notion of logic and ambitious enough requirements for a framework, there is in principle no guarantee that such an $\mathcal{F}$ will necessarily be found. However, somewhat more restricted successes such as finding an $\mathcal{F}$ in which all the logics of "practical interest," having finitary presentations of their syntax and their rules, can be represented can be very valuable and can provide a great economy of effort. This is because, if an implementation for such a framework logic exists, it becomes possible to implement through it all the other "object logics" that can be adequately represented in the framework logic.

Much work has already been done in this area, including the Edinburgh logical framework LF [25,26,17] and meta-theorem-provers such as Isabelle [51], λProlog [49,16], and Elf [52], all of which adopt as framework logics different variants of higher-order logics or type theories. There has also been important work on what Basin and Constable [3] call *metalogical* frameworks. These are frameworks supporting reasoning about the metalogical aspects of the logics being represented. Typically, this is accomplished by reifying as "data" the proof theory of the logic being represented in a process that is described in [3] as *externalizing* the logic in question. This is in contrast to the more *internalized* form in which logics are represented in LF and in meta-theorem-provers, so that deduction in the object logic is mirrored by deduction—for example, type inference—in the framework logic. Work on metalogical frameworks includes the already mentioned paper by Basin and Constable [3], who advocate constructive type theory as the framework logic, work of Matthews, Smaill and Basin [38], who use Feferman's FS_0 [15], a logic designed with the explicit purpose of being a metalogical framework, earlier work by Smullyan [56], and work by Goguen, Stevens, Hobley and Hilberdink [23] on the 2OBJ meta-theorem-prover, which uses order-sorted equational logic [22,24].

A difficulty with systems based on higher-order type theory such as LF is that it may be quite awkward and of little practical use to represent logics whose structural properties differ considerably from those of the type theory. For example, linear and relevance logics do not have adequate representations in LF, in a precise technical sense of "adequate" [17, Corollary 5.1.8]. Since in metalogical frameworks a direct connection between deduction in the object and framework logics does not have to be maintained, they seem in principle much more flexible in their representational capabilities. However, this comes at a price, since the possibility of directly using an implementation of the framework logic to implement an object logic is compromised.

In relation to this previous work, rewriting logic seems to have great flexibility to represent in a natural way many other logics, widely different in nature, including equational, Horn, and linear logics, and any sequent calculus presentation of a logic under extremely general assumptions about such a logic. Moreover, quantifiers can also be treated without problems. More experience in representing other logics is certainly needed, but we are encouraged by the naturalness and directness—often preserving the original syntax and rules—with which the logics that we have studied can be represented. This is due to the great simplicity and generality of rewriting logic, since in it all syntax and structural axioms are user-definable, so that the abstract syntax of an object logic can be represented as an algebraic data type, and is also due to the existence of only a few general "meta" rules of deduction relative to the rewrite rules given by a specification, where such a specification can be used to describe with rewrite rules the rules of deduction of the object logic in question. In addition, the direct correspondence between proofs in object logics and proofs in the framework logic can often be maintained in a *conservative* way by means of maps of logics, so that an implementation of rewriting logic can directly support an implementation of an object logic. Furthermore, given the directness with which logics can be represented, the task of proving conservativity is in many cases straightforward. Finally, although we do not discuss this aspect, externalization of logics to support metalogical reasoning is also possible in rewriting logic [37].

Another important difference is that most approaches to logical frameworks are proof-theoretic in nature, and thus they do not address the model theories of the logics being represented. By contrast, several of the representations into rewriting logic that we have studied—such as those for equational logic, Horn logic, and linear logic—involve both models and proofs and are therefore considerably more informative than purely proof-theoretic representations.

As we have already mentioned, the distinction between a logical system and a language or a model of computation is more and more in the eyes of the beholder, although of course efficiency considerations and the practical uses intended may indeed strongly influence the design choices. Therefore, even though at the most basic mathematical level there may be little distinction between the general way in which a logic, a programming language, a system, or a model of computation are represented in rewriting logic, the criteria and case studies to be used in order to judge the merits of rewriting logic as a semantic framework are different from those relevant for its use as a logical framework.

One important consideration is that, from a computational point of view, rewriting logic deduction is intrinsically *concurrent*. In fact, it was the search for a general concurrency model that would help unify the somewhat bewildering heterogeneity of existing models that provided the original impetus for the first investigations on rewriting logic [40]. The generality and naturalness with which many concurrency models can be expressed in rewriting logic has already been illustrated at length in several papers [40,42]. In this paper, we just discuss in some detail the case of Milner's CCS [48].

Deduction with constraints can greatly increase the efficiency of theorem provers and logic programming languages. The most classical constraint solving algorithm is syntactic unification, which corresponds to solving equations in a free algebra, the so-called Herbrand model, and is used in resolution. However, much more efficient deduction techniques than those afforded by resolution can be obtained by building in additional knowledge of special theories in the form of constraint solving algorithms such as, for example, semantic unification, or equalities and inequalities in a numerical domain. In the past few years many authors have become aware that many constraint solving algorithms can be specified declaratively using rewrite rules. However, since constraint solving is usually nondeterministic, the usual equational logic interpretation of rewrite rules is clearly inadequate as a mathematical semantics. By contrast, rewriting logic completely avoids such inadequacies and can serve as a semantic framework for logical systems and languages using constraints, including parallel ones.

The paper begins with a summary of the theory of general logics [39] that provides the conceptual basis for our discussion of logical frameworks. Then the rules of deduction of rewriting logic are introduced, and the Maude language based on rewriting logic is briefly discussed. This is followed by three sections illustrating the representation of logics in the rewriting logic framework: linear logic, quantifiers and sequent systems. Using the fact that rewriting logic is reflective [9,10], it is often possible to reify inside rewriting logic itself a representation map $\mathcal{L} \rightarrow \mathit{RWLogic}$ for the finitely presentable theories of $\mathcal{L}$. Such a reification takes the form of a map between the abstract data types representing the finitary theories of $\mathcal{L}$ and of *RWLogic*, as we illustrate with the linear logic example. The use of rewriting logic as a semantic framework is illustrated by means of the CCS and constraint solving examples. The paper ends with some concluding remarks.

2 General logics

A modular and general axiomatic theory of logics should adequately cover all the key ingredients of a logic. These include: a *syntax*, a notion of *entailment* of a sentence from a set of axioms, a notion of *model*, and a notion of *satisfaction* of a sentence by a model. The theory of *general logics* [39] provides axiomatic notions formalizing the different aspects of a logic and of their combinations into fuller notions of logic:

- An *entailment system* axiomatizes the consequence relation of a logic.
- The notion of *institution* [19,20] covers the model-theoretic aspects of a logic, focusing on the notion of satisfaction.
- A *logic* is obtained by combining an entailment system and an institution.
- A *proof calculus* enriches an entailment system with an actual proof theory.
- A *logical system* is a logic with a choice of a proof calculus for it.

Here we give a brief review of the required notions; a detailed account with many examples can be found in [39] (see also [36,37]).

Syntax can typically be given by a *signature* Σ providing a grammar on which to build *sentences*. We assume that for each logic there is a category **Sign** of possible signatures for it, and a functor *sen* assigning to each signature Σ the set $sen(\Sigma)$ of all its sentences. For a given signature Σ in **Sign**, *entailment* (also called *provability*) of a sentence $\varphi \in sen(\Sigma)$ from a set of axioms $\Gamma \subseteq sen(\Sigma)$ is a relation $\Gamma \vdash \varphi$ which holds if and only if we can prove φ from the axioms Γ using the rules of the logic. This relation must be reflexive, monotonic, transitive, and must preserve translations between signatures. These components constitute an *entailment system.*

A *theory* in a given entailment system is a pair (Σ, Γ) with Σ a signature and $\Gamma \subseteq sen(\Sigma)$.

An institution consists of a category **Sign** of signatures and a functor $sen : \mathbf{Sign} \to \mathbf{Set}$ associating to each signature a set of sentences, together with a functor **Mod** that associates to each signature Σ a category of Σ-*models*, and a binary relation $\models$ between models and sentences called *satisfaction* satisfying appropriate requirements.

Combining an entailment system and an institution we obtain a *logic*, defined as a 5-tuple $\mathcal{L} = (\mathbf{Sign}, sen, \mathbf{Mod}, \vdash, \models)$ such that:

- $(\mathbf{Sign}, sen, \vdash)$ is an entailment system,
- $(\mathbf{Sign}, sen, \mathbf{Mod}, \models)$ is an institution,

and the following *soundness condition* is satisfied: for any $\Sigma \in |\mathbf{Sign}|$, $\Gamma \subseteq sen(\Sigma)$, and $\varphi \in sen(\Sigma)$,

$$\Gamma \vdash \varphi \implies \Gamma \models \varphi,$$

where, by definition, the relation $\Gamma \models \varphi$ holds if and only if $M \models \varphi$ holds for any model M that satisfies all the sentences in Γ.

The detailed treatment in [39] includes also a flexible axiomatic notion of a *proof calculus*—in which proofs of entailments, not just the entailments themselves, are first class citizens—and the notion of a *logical system* that consists of a logic together with a choice of a proof calculus for it.

One of the most interesting fruits of the theory of general logics is that it gives us a method for *relating* logics in a general and systematic way, and to exploit such relations in many applications. The key notion is that of a *mapping* translating one logic into another

$$\mathcal{L} \longrightarrow \mathcal{L}'$$

that preserves whatever logical properties are relevant, such as provability of formulas, or satisfaction of a formula by a model. Therefore, we have maps of entailment systems, institutions, logics, proof calculi, and logical systems. Such mappings allow us to relate in a rigorous way different logics, to combine different formalisms together, and to explore new logics for computational purposes. A detailed treatment of such maps is given in [39]; here we just give a brief sketch.

Basically, a map of entailment systems $(\Phi, \alpha) : \mathcal{E} \longrightarrow \mathcal{E}'$ maps the language of $\mathcal{E}$ to that of $\mathcal{E}'$ in a way that respects the entailment relation. This means that signatures of $\mathcal{E}$ are functorially mapped by Φ to signatures of $\mathcal{E}'$, and that sentences of $\mathcal{E}$ are mapped by α to sentences of $\mathcal{E}'$ in a way that is coherent with the mapping of their corresponding signatures. In addition, α must respect the entailment relations $\vdash$ of $\mathcal{E}$ and $\vdash'$ of $\mathcal{E}'$, i.e., we must have

$$\Gamma \vdash \varphi \;\Rightarrow\; \alpha(\Gamma) \vdash' \alpha(\varphi).$$

The map is *conservative* when this implication is an equivalence. For many interesting applications one needs to map signatures of $\mathcal{E}$ to *theories* of $\mathcal{E}'$, that is, Σ is mapped by Φ to (Σ', Γ'), with $\Gamma' \subseteq sen'(\Sigma')$. It is this more general notion of *map between entailment systems* that is axiomatized by the definition in [39].

A *map of institutions* $(\Phi, \alpha, \beta) : \mathcal{I} \longrightarrow \mathcal{I}'$ is similar in its syntax part to a map of entailment systems. In addition, for models we have a natural functor $\beta : \mathbf{Mod}'(\Phi(\Sigma)) \longrightarrow \mathbf{Mod}(\Sigma)$ "backwards" from the models in $\mathcal{I}'$ of a translated signature $\Phi(\Sigma)$ to the models in $\mathcal{I}$ of the original signature Σ, and such a mapping respects the satisfaction relations $\models$ of $\mathcal{I}$ and $\models'$ of $\mathcal{I}'$, in the sense that $M' \models' \alpha(\varphi) \iff \beta(M') \models \varphi$. Maps of institutions are different from the *institution morphisms* in [20].

A *map of logics* has now a very simple definition. It consists of a pair of maps: one for the underlying entailment systems, and another for the underlying institutions, such that both maps agree on how they translate signatures and sentences. There are also notions of *map of proof calculi* and *map of logical systems*, for which we refer the reader to [39].

As we have already explained in the introduction, viewed from the perspective of a general space of logics that can be related to each other by means of mappings, the quest for a *logical framework* can be understood as the search within such a space for a logic $\mathcal{F}$ (the *framework* logic) such that many other logics (the *object* logics) such as, say, $\mathcal{L}$ can be represented in $\mathcal{F}$ by means of mappings $\mathcal{L} \longrightarrow \mathcal{F}$ that have good enough properties. The minimum requirement that seems reasonable to make on a representation map $\mathcal{L} \longrightarrow \mathcal{F}$ is that it should be a *conservative* map of entailment systems. Under such circumstances, we can reduce issues of provability in $\mathcal{L}$ to issues of provability in $\mathcal{F}$, by mapping the theories and sentences of $\mathcal{L}$ into $\mathcal{F}$ using the conservative representation map. Given a computer implementation of deduction in $\mathcal{F}$, we can use the conservative map to prove theorems in $\mathcal{L}$ by proving the corresponding translations in $\mathcal{F}$. In this way, the implementation for $\mathcal{F}$ can be used as a generic theorem-prover for many logics.

However, since maps between logics can, as we have seen, respect additional logical structure such as the model theory or the proofs, in some cases a representation map into a logical framework may be particularly informative because, in addition to being a conservative map of entailment systems, it is also a map of institutions, or a map of proof calculi. For example, when rewriting logic is chosen as a logical framework, appropriate representation maps for equational logic, Horn logic, and propositional linear logic can be

shown to be maps of institutions also [36]. In general, however, since the model theories of different logics can be very different from each other, it is not reasonable to expect or require that the representation maps into a logical framework will always be maps of institutions. Nevertheless, what it can always be done is to "borrow" the additional logical structure that $\mathcal{F}$ may have (institution, proof calculus) to endow $\mathcal{L}$ with such a structure, so that the representation map does indeed preserve the extra structure [7].

Having criteria for the adequacy of maps representating logics in a logical framework is not enough. An equally important issue is having criteria for the *generality* of a logical framework, so that it is in fact justified to call it by that name. That is, given a candidate logical framework $\mathcal{F}$, how many logics can be adequately represented in $\mathcal{F}$? We can make this question precise by defining the *scope* of a logical framework $\mathcal{F}$ as the class of entailment systems $\mathcal{E}$ having conservative maps of entailment systems $\mathcal{E} \longrightarrow \mathcal{F}$. In this regard, the axioms of the theory of general logics that we have presented are probably too general; without adding further assumptions it is not reasonable to expect that we can find a logical framework $\mathcal{F}$ whose scope is the class of *all* entailment systems. A much more reasonable goal is finding an $\mathcal{F}$ whose scope includes all entailment systems of "practical interest," having finitary presentations of their syntax and their rules of deduction. Axiomatizing such finitely presentable entailment systems and proof calculi so as to capture—in the spirit of the more general axioms that we have presented, but with stronger requirements—all logics of "practical interest" (at least for computational purposes) is a very important research task.

Another important property that can help measuring the suitability of a logic $\mathcal{F}$ as a logical framework is its *representational adequacy*, understood as the naturalness and ease with which entailment systems can be represented, so that the representation $\mathcal{E} \longrightarrow \mathcal{F}$ mirrors $\mathcal{E}$ as closely as possible. That is, a framework requiring very complicated encodings for many object logics of interest is less representationally adequate than one for which most logics can be represented in a straightforward way, so that there is in fact little or no "distance" between an object logic and its corresponding representation. Although at present we lack a precise definition of this property, it is quite easy to observe its absence in particular examples. We view representational adequacy as a very important practical criterion for judging the relative merits of different logical frameworks.

In this paper, we present rewriting logic as a logic that seems to have particularly good properties as a logical framework. The evidence we can give within the space constraints of this paper is necessarily partial; further evidence can be found in [36,37]. We conjecture that the scope of rewriting logic contains all entailment systems of "practical interest" for a reasonable axiomatization of such systems.

3 Rewriting logic

A *signature* in rewriting logic is an equational theory (Σ, E), where Σ is an equational signature and E is a set of Σ-equations[3]. Rewriting will operate on equivalence classes of terms modulo E. In this way, we free rewriting from the syntactic constraints of a term representation and gain a much greater flexibility in deciding what counts as a *data structure*; for example, string rewriting is obtained by imposing an associativity axiom, and multiset rewriting by imposing associativity and commutativity. Of course, standard term rewriting is obtained as the particular case in which the set of equations E is empty. Techniques for rewriting modulo equations have been studied extensively [14] and can be used to implement rewriting modulo many equational theories of interest.

Given a signature (Σ, E), *sentences* of rewriting logic are of the form

$$[t]_E \longrightarrow [t']_E,$$

where t and t' are Σ-terms possibly involving some variables, and $[t]_E$ denotes the equivalence class of the term t modulo the equations E. A theory in this logic, called a *rewrite theory*, is a slight generalization of the usual notion of theory in that, in addition, we allow the axioms $[t]_E \longrightarrow [t']_E$ (called *rewrite rules*) to be labelled, because this is very natural for many applications[4].

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sentence $[t] \longrightarrow [t']$, or that $[t] \longrightarrow [t']$ is a *(concurrent) $\mathcal{R}$-rewrite*, and write $\mathcal{R} \vdash [t] \longrightarrow [t']$ if and only if $[t] \longrightarrow [t']$ can be obtained by finite application of the following *rules of deduction* (where we assume that all the terms are well formed and $t(\overline{w}/\overline{x})$ denotes the simultaneous substitution of w_i for x_i in t):

(i) **Reflexivity**. For each $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] \longrightarrow [t]}$.

(ii) **Congruence**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{[t_1] \longrightarrow [t'_1] \quad \ldots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}.$$

(iii) **Replacement**. For each rule $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R,

$$\frac{[w_1] \longrightarrow [w'_1] \quad \ldots \quad [w_n] \longrightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}.$$

(iv) **Transitivity**

$$\frac{[t_1] \longrightarrow [t_2] \quad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}.$$

---

[3] Rewriting logic is parameterized with respect to the version of the underlying equational logic, which can be unsorted, many-sorted, order-sorted, or the recently developed membership equational logic [6,43].

[4] Moreover, the main results of rewriting logic have been extended to conditional rules in [40] with very general rules of the form

$$r : [t] \longrightarrow [t'] \ \textit{if} \ [u_1] \longrightarrow [v_1] \wedge \ldots \wedge [u_k] \longrightarrow [v_k].$$

This increases considerably the expressive power of rewrite theories, of which later examples in this paper make use.

Rewriting logic is a logic for reasoning correctly about *concurrent systems* having *states*, and evolving by means of *transitions*. The signature of a rewrite theory describes a particular structure for the states of a system—e.g., multiset, binary tree, etc.—so that its states can be distributed according to such a structure. The rewrite rules in the theory describe which *elementary local transitions* are possible in the distributed state by concurrent local transformations. The rules of rewriting logic allow us to reason correctly about which *general* concurrent transitions are possible in a system satisfying such a description. Thus, computationally, each rewriting step is a parallel local transition in a concurrent system.

Alternatively, however, we can adopt a logical viewpoint instead, and regard the rules of rewriting logic as *metarules* for correct deduction in a *logical system*. Logically, each rewriting step is a logical *entailment* in a formal system. This second viewpoint is particularly fruitful when using rewriting logic as a logical framework [36,37].

Thus, in rewriting logic $[t]$ should not be understood as a *term* in the usual first-order logic sense, but as a *proposition* or *formula*—built up using the *connectives* in Σ—that asserts being in a certain *state* having a certain *structure*. However, unlike most other logics, the logical connectives Σ and their structural properties E are entirely *user-definable*. This provides great flexibility for considering many different state structures and makes rewriting logic very general in its capacity to deal with many different types of concurrent systems, and also in its capacity to represent many different logics.

The computational and the logical viewpoints under which rewriting logic can be interpreted can be summarized in the following diagram of correspondences, which will be further illustrated by means of the examples described in later sections.

State	$\longleftrightarrow$	*Term*	$\longleftrightarrow$	*Proposition*
Transition	$\longleftrightarrow$	*Rewriting*	$\longleftrightarrow$	*Deduction*
Distributed Structure	$\longleftrightarrow$	*Algebraic Structure*	$\longleftrightarrow$	*Propositional Structure*

The model theory of rewriting logic has been developed in detail in [40], where initiality, soundness and completeness theorems are proved. Rewriting logic models, called $\mathcal{R}$*-systems*, have a natural *category* structure, with states (or formulas) as objects, transitions (or proofs) as morphisms, and sequential composition as morphism composition, and in them dynamic behavior exactly corresponds to deduction.

In what follows, we will use the syntax of Maude [41,46,8], a wide spectrum programming language directly based on rewriting logic, to present rewrite theories. In Maude, there are essentially two kinds of *modules*: [5]

- *Functional modules*, which are of the form `fmod` $\mathcal{E}$ `endfm` for an equational

[5] There are also *object-oriented modules*, which can be reduced to a special case of system modules [41] and are not used here.

theory $\mathcal{E}$, and

- *System modules*, which are of the form `mod` $\mathcal{R}$ `endm` for a rewrite theory $\mathcal{R}$.

In functional modules, equations are declared with the keywords `eq` or `ceq` (for conditional equations), and in system modules with the keywords `ax` or `cax`. Certain equations, such as associativity, commutativity, or identity, for which rewriting modulo is provided, can be declared together with the corresponding function using the keywords `assoc`, `comm`, `id`. Rules can only appear in system modules, and are declared with the keywords `rl` or `crl`.

The version of rewriting logic used for Maude in this paper is *order-sorted*[6]. This means that rewrite theories are typed (types are called *sorts*) and can have subtypes (subsorts), and that function symbols can be overloaded. In particular, functional modules are order-sorted equational theories [22] and they form a sublanguage similar to OBJ [24]. Logically, this corresponds to a map of logics

$$OSEqtl \longrightarrow OSRWLogic$$

embedding (order-sorted) equational logic within (order-sorted) rewriting logic. The details of this map of logics are discussed in [36, Section 4.1].

As in OBJ, Maude modules can be imported by other modules, and can also be parameterized by means of *theories* that specify semantic requirements for interfaces.

Since the power and the range of applications of a multiparadigm logic programming language can be substantially increased if it is possible to solve queries involving *logical variables* in the sense of relational programming, as in the Prolog language, we are naturally led to seek a unification of the three paradigms of functional, relational and concurrent object-oriented programming into a single multiparadigm logic programming language. This unification can be attained in a language extension of Maude called MaudeLog. The integration of Horn logic is achieved by a map of logics

$$OSHorn \longrightarrow OSRWLogic$$

that systematically relates order-sorted Horn logic to order-sorted rewriting logic. The details of this map are discussed in [36, Section 4.2].

4 Linear logic

In this section, we describe a map of logics $LinLogic \longrightarrow OSRWLogic$ mapping theories in full quantifier-free first-order linear logic to rewrite theories. We do not provide much motivation for linear logic, referring the reader to [18,57,35] for example. We need to point out, nonetheless, the way linear logic is seen as an entailment system. If one thinks of formulas as sentences and of the turnstile symbol "$\vdash$" in a sequent as the entailment relation, then this relation is not monotonic, because in linear logic the structural rules of weakening

[6] The latest version of Maude [8] is based on the recently developed membership equational logic, which extends order-sorted equational logic and at the same time has a simpler and more general model theory [6,43].

and contraction are forbidden, so that, for example, we have the sequent $A \vdash A$ as an axiom, but we cannot derive either $A, B \vdash A$ or even $A, A \vdash A$. The point is that, for Σ a linear logic signature, the elements of $sen(\Sigma)$ should not be identified with *formulas* but with *sequents*. Viewed as a way of generating sequents, i.e., identifying our entailment relation $\vdash$ with the closure of the horizontal bar relation among linear logic sequents, the entailment of linear logic is indeed reflexive, monotonic and transitive. This idea is also supported by the categorical models for linear logic [55,35], in which sequents are interpreted as morphisms, and leads to a very natural correspondence between the models of rewriting and linear logic.

We use the syntax of the Maude language to write down the map of entailment systems from linear logic to rewriting logic. Note that any sequence of characters starting with either "---" or "***" and ending with "end-of-line" is a comment. Also, from now on, we usually drop the equivalence class square brackets, adopting the convention that a term t denotes the equivalence class $[t]_E$ for the appropriate set of structural axioms E.

We first define the *functional* theory `PROPO[X]` which introduces the syntax of propositions as a parameterized abstract data type. The parameterization permits having additional structure at the level of atoms if desired. In order to provide a proper treatment of negation, only equations are given, and no rewrite rules are introduced in this theory; they are introduced afterwards in the `LINLOG[X]` theory. The purpose of the equations in the `PROPO[X]` theory is to push negation to the atom level, by using the dualities of linear logic; this is a well-known process in classical and linear logic.

```
fth ATOM is
  sort Atom .
endft

fth PROPO[X :: ATOM] is
  sort Prop0 .
  subsort Atom < Prop0 .
  ops 1 0 ⊥ ⊤ : -> Prop0 .
  op _⊥ : Prop0 -> Prop0 .
  op _⊗_ : Prop0 Prop0 -> Prop0 [assoc comm id: 1] .
  op _⅋_ : Prop0 Prop0 -> Prop0 [assoc comm id: ⊥] .
  op _⊕_ : Prop0 Prop0 -> Prop0 [assoc comm id: 0] .
  op _&_ : Prop0 Prop0 -> Prop0 [assoc comm id: ⊤] .
  op !_ : Prop0 -> Prop0 .
  op ?_ : Prop0 -> Prop0 .

  vars A B : Prop0 .
  eq (A ⊗ B)⊥ = A⊥ ⅋ B⊥ .
  eq (A ⅋ B)⊥ = A⊥ ⊗ B⊥ .
  eq (A & B)⊥ = A⊥ ⊕ B⊥ .
  eq (A ⊕ B)⊥ = A⊥ & B⊥ .
  eq (!A)⊥ = ?(A⊥) .
```

```
  eq (?A)^⊥ = !(A^⊥) .
  eq A^⊥^⊥ = A .
  eq 1^⊥ = ⊥ .
  eq ⊥^⊥ = 1 .
  eq ⊤^⊥ = 0 .
  eq 0^⊥ = ⊤ .
endft
```

The `LINLOG[X]` theory introduces linear logic propositions and the rules of the logic. Propositions are of the form `[A]` for `A` an expression in `Prop0`. All logical connectives work similarly for `Prop0` expressions and for propositions, except negation, which is defined only for `Prop0` expressions.

Some presentations of linear logic are given in the form of one-sided sequents $\vdash \Gamma$ where negation has been pushed to the atom level, and there are no rules for negation in the sequent calculus [18]. In this section, in order to make the connections with category theory and with rewriting logic more direct, we prefer to use standard sequents of the more general form $\Gamma \vdash \Delta$. In Section 6, we will also use one-sided sequents just in order to reduce the number of rules.

The style of our formulation adopts a categorical viewpoint for the proof theory and semantics of linear logic [55,35]). This style exploits the close connection between the models of linear logic and those of rewriting logic which are also categories, as we have mentioned in Section 3. When seeking the minimal categorical structure required for interpreting linear logic, an important question is how to interpret the connective $\parr$ without using negation, and how to axiomatize its relationship with the tensor $\otimes$. Cockett and Seely have answered this question with the notion of a *weakly distributive category* [11]. A weakly distributive category consists of a category $\mathcal{C}$ with two symmetric tensor products $\otimes, \parr\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, and a natural transformation $A \otimes (B \parr C) \longrightarrow (A \otimes B) \parr B$ (weak distributivity) satisfying some coherence equations[7]. Negation is added to a weakly distributive category by means of a function $(_)^{\perp} : |\mathcal{C}| \to |\mathcal{C}|$ on the objects of $\mathcal{C}$, and natural transformations $1 \longrightarrow A \parr A^{\perp}$ and $A \otimes A^{\perp} \longrightarrow \perp$ satisfying some coherence equations.

In the following theory, the rewrite rules for $\otimes$, $\parr$ and negation correspond to the natural transformations in the definition of a weakly distributive category, as explained above. The rules for & ($\oplus$, respectively) mirror the usual definition of final object and product (initial object and coproduct, respectively). Finally, the axioms and rules for the exponential ! (?, respectively) correspond to a comonad with a comonoid structure (monad with monoid structure, respectively). Note that some rules are redundant, but we have decided to include them in order to make the connectives less interdependent, so that, for example, if the connective & is omitted we do not need to add new rules for the modality !.

[7] Cockett and Seely develop in [11] the more general case in which the tensor products are not assumed to be symmetric.

```
th LINLOG[X :: ATOM] is
  protecting PROP0[X] .
  sort Prop .
  ops 1 0 ⊥ ⊤ : -> Prop .
  op _⊗_ : Prop Prop -> Prop [assoc comm id: 1] .
  op _⅋_ : Prop Prop -> Prop [assoc comm id: ⊥] .
  op _⊕_ : Prop Prop -> Prop [assoc comm id: 0] .
  op _&_ : Prop Prop -> Prop [assoc comm id: ⊤] .
  op !_ : Prop -> Prop .
  op ?_ : Prop -> Prop .

  op [_] : Prop0 -> Prop .

  vars A B : Prop0 .
  ax [A ⊗ B] = [A] ⊗ [B] .
  ax [A ⅋ B] = [A] ⅋ [B] .
  ax [A & B] = [A] & [B] .
  ax [A ⊕ B] = [A] ⊕ [B] .
  ax [!A] = ![A] .
  ax [?A] = ?[A] .
  ax [1] = 1 .
  ax [⊥] = ⊥ .
  ax [⊤] = ⊤ .
  ax [0] = 0 .

  *** [_] is injective
  cax A = B if [A] = [B] .

  *** Rules for negation
  rl 1 => [A] ⅋ [A^⊥] .
  rl [A] ⊗ [A^⊥] => ⊥ .

  vars P Q R : Prop .
  *** Rules for ⊗ and ⅋
  rl P ⊗ (Q ⅋ R) => (P ⊗ Q) ⅋ R .

  *** Rules for &
  rl P => ⊤ .                                    *** (1)
  rl P & Q => P .
  crl R => P & Q  if  R => P  and  R => Q .  *** (2)

  *** Rules for ⊕
  rl 0 => P .                                     *** (3)
  rl P => P ⊕ Q .
  crl P ⊕ Q => R  if  P => R  and  Q => R .  *** (4)
```

```
  *** Structural axioms and rules for !
  ax !(P & Q) = !P ⊗ !Q .                          *** (5)
  ax !⊤ = 1 .                                      *** (6)

  rl !P => P .
  rl !P => !!P .
  rl !P => 1 .          *** redundant from (1) and (6) above
  rl !P => !P ⊗ !P .    *** redundant from (2) and (5) above

  *** Structural axioms and rules for ?
  ax ?(P ⊕ Q) = ?P ⅋ ?Q .                          *** (7)
  ax ?0 = ⊥ .                                      *** (8)

  rl P => ?P .
  rl ??P => ?P .
  rl ⊥ => ?P .          *** redundant from (3) and (8) above
  rl ?P ⅋ ?P => ?P .    *** redundant from (4) and (7) above
endt
```

A linear logic formula is built from a set of propositional constants using the logical constants and connectives of linear logic. Notice that linear implication $A \multimap B$ is not necessary because it can be defined as $A^{\perp} \parr B$.

A *linear theory* T in propositional linear logic consists of a finite set C of propositional constants and a finite set of sequents of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$, where each A_i and B_j is a linear logic formula built from the constants in C. Given such a theory T, it is interpreted in rewriting logic as follows.

First, we define a functional theory to interpret the propositional constants in C. For example, if $C = \{a, b, c\}$ we would define

```
fth C is
  sort Atom .
  ops a b c : -> Atom .
endft
```

Then, we can instantiate the parameterized theory `LINLOG[X]` using this functional theory, with the default view `ATOM` $\rightarrow$ `C`:

```
make LINLOG0 is LINLOG[C] endmk
```

A linear logic formula A (with constants in C) is interpreted in `LINLOG0` as the term `[A]` of sort `Prop`. For example, the formula $(a \otimes b)^{\perp} \oplus (!(a \,\&\, c^{\perp}))^{\perp}$ is interpreted as the term

```
[(a ⊗ b)^⊥ ⊕ (!(a & c^⊥))^⊥]
```

which is equal to the following term, using the equations in `PROP0[X]` and the structural axioms in `LINLOG[X]`,

```
([a^⊥] ⅋ [b^⊥]) ⊕ ?([a^⊥] ⊕ [c]).
```

Finally, we extend the theory LINLOG0 by adding a rule

```
rl [A1] ⊗ ... ⊗ [An] => [B1] ⅋ ... ⅋ [Bm] .
```

for each sequent $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ in the linear theory T. For example, if T consists of the two sequents

$$a \otimes b, !c \oplus a \vdash a, (c \oplus b)^{\perp}$$

$$a \parr b, ?(c^{\perp}) \vdash (?b \parr !c)^{\perp}, a \oplus b,$$

the corresponding rewrite theory is

```
th LINLOG(T) is
  extending LINLOG0 .
  rl [a] ⊗ [b] ⊗ (![c] ⊕ [a]) => [a] ⅋ ([c⊥] & [b⊥]) .
  rl ([a] ⅋ [b]) ⊗ ?[c⊥] => (![b⊥] ⊗ ?[c⊥]) ⅋ ([a] ⊕ [b]) .
endt
```

The main result is the following conservativity theorem:

Theorem 4.1 *Given a linear theory T, a sequent $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ is provable in linear logic from the axioms in T if and only if the sequent*

```
[A1] ⊗ ... ⊗ [An] ⟶ [B1] ⅋ ... ⅋ [Bm]
```

is a LINLOG(T)*-rewrite, i.e., it is provable in rewriting logic from the rewrite theory* LINLOG(T)*.*

Thus, we have a map of entailment systems between linear logic and rewriting logic, which is *conservative*. Moreover, Section 4.3.3 of [36] explains in detail how to obtain from a model of the rewriting logic theory LINLOG(T) a model of the linear logic theory T, in such a way as to extend this map of entailment systems to a *conservative map of logics* $\Phi : \mathit{LinLogic} \longrightarrow \mathit{OSRWLogic}$.

5 Quantifiers

In Section 4 we have defined a map of logics between quantifier-free linear logic and rewriting logic. In this section, we show a technique that can be used to extend that map at the level of entailment systems to quantifiers. Our equational treatment of quantification, inspired by ideas of Laneve and Montanari on the definition of the lambda calculus as a theory in rewriting logic [33], is very general and encompasses not only existential and universal quantification, but also lambda abstraction and other such binding mechanisms.

The main idea is to internalize as operations in the theory the notions of free variables and substitution that are usually defined at the metalevel. Then, the typical definitions of such notions by structural induction on terms can be easily written down as equations in the theory, but, more importantly, we can consider terms modulo these axioms and we can also use the operation of substitution explicitly in the rules introducing or eliminating quantifiers. This is similar to the lambda calculus with explicit substitutions defined by Abadi, Cardelli, Curien, and Lévy in [1].

We only present here the example of the lambda abstraction binding mechanism in the lambda calculus, as defined by Laneve and Montanari in [33]. We assume a parameterized functional module SET[X] that provides finite sets over a parameter set X with operations _U_ for union, _-_ for set difference, {_} for singleton, emptyset for the empty set, and a predicate _is-in_ for membership.

```
fth VAR is
  sort Var .
  protecting SET[Var] .
  op new: Set -> Var .
  var S : Set .
  eq new(S) is-in S = false .  *** new variable
endft

fmod LAMBDA[X :: VAR] is
  extending SET[X] .
  sort Lambda .
  subsort Var < Lambda .                          *** variables
  op λ_._ : Var Lambda -> Lambda .       *** lambda abstraction
  op __ : Lambda Lambda -> Lambda .               *** application
  op _[_/_] : Lambda Lambda Var -> Lambda .       *** substitution
  op fv : Lambda -> Set .                         *** free variables

  vars X Y : Var .
  vars M N P : Lambda .

  *** Free variables
  eq fv(X) = {X} .
  eq fv(λX.M) = fv(M) - {X} .
  eq fv(MN) = fv(M) U fv(N) .
  eq fv(M[N/X]) = (fv(M) - {X}) U fv(N) .

  *** Substitution equations
  eq X[N/X] = N .
  ceq Y[N/X] = Y  if  not(X == Y) .
  eq (MN)[P/X] = (M[P/X])(N[P/X]) .
  eq (λX.M)[N/X] = λX.M .
  ceq (λY.M)[N/X] = λY.(M[N/X])   if not(X == Y) and
                (not(Y is-in fv(N)) or not(X is-in fv(M))) .
  ceq (λY.M)[N/X] = λ(new(fv(MN))).((M[new(fv(MN))/Y])[N/X])
       if not(X == Y) and Y is-in fv(N) and X is-in fv(M) .
endfm
```

Note that substitution is here another term constructor instead of a meta-syntactic operation. Of course, using the above equations, all occurrences of the substitution constructor can be eliminated. After having defined in the previous functional module the class of lambda terms with substitution, we

just need to add the equational axiom of alpha conversion and the beta rule in the following module:

```
mod ALPHA-BETA[X :: VAR] is
  extending LAMBDA[X] .
  vars X Y : Var .
  vars M N : Lambda .

  *** Alpha conversion
  cax λX.M = λY.(M[Y/X])  if  not(Y is-in fv(M)) .

  *** Beta reduction
  rl (λX.M)N => M[N/X] .
endm
```

In order to introduce quantifiers, we can develop a completely similar approach, by first introducing substitution in the syntax together with the quantifiers, and then adding rewrite rules for the new connectives. A detailed application of this technique to first-order linear logic is developed in Section 4.4 of [36].

6 Sequent systems

In Section 4, we have mapped linear logic formulas to terms, and linear logic sequents to rewrite rules in rewriting logic. There is another map of entailment systems between linear logic and rewriting logic in which linear sequents become also terms, and rewrite rules correspond to rules in a Gentzen sequent calculus for linear logic. In order to reduce the number of rules of this calculus, we consider one-sided linear sequents in this section, but a completely similar treatment can be given for two-sided sequents. Thus, a linear logic sequent will be a turnstile symbol "$\vdash$" followed by a multiset M of linear logic formulas, that in our translation to rewriting logic will be represented by the term `⊢ M`. Using the duality of linear logic negation, a two-sided sequent $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ can in this notation be expressed as the one-sided sequent $\vdash A_1^{\perp}, \ldots, A_n^{\perp}, B_1, \ldots, B_m$.

First, we define a parameterized module for multisets.

```
fth ELEM is
  sort Elem .
endft

fmod MSET[X :: ELEM] is
  sort Mset .
  subsort Elem < Mset .
  op null : -> Mset .
  op _,_ : Mset Mset -> Mset [assoc comm id: null] .
endfm
```

Now we can use this parameterized module to define the main module for

sequents[8] and give the corresponding rules. A sequent calculus rule of the form

$$\frac{\vdash M_1, \ldots, \vdash M_n}{\vdash M}$$

becomes the rewrite rule

```
rl ⊢ M1 ... ⊢ Mn => ⊢ M .
```

on the sort `Configuration`. Recalling that "---" introduces a comment, this rule can be written as

```
rl    ⊢ M1 ... ⊢ Mn
   => --- ----------
          ⊢ M .
```

In the module below, we assume a module `FO-PROP0[X]` extending the syntax of propositional linear logic in `PROP0[X]` to the syntax of first-order linear logic, by means of the technique described in Section 5. See [36, Section 4.4] for the complete details.

```
mod LL-SEQUENT[X :: VAR] is
  protecting FO-PROP0[X] .
  extending MSET[FO-PROP0[X]] .
  --- a configuration is a multiset of sequents
  sort Configuration .
  op ⊢_ : Mset -> Configuration .
  op empty : -> Configuration .
  op __ : Configuration Configuration -> Configuration
                                        [assoc comm id: empty] .
  op ?_ : Mset -> Mset .
  vars M N : Mset .
  ax ?null = null .
  ax ?(M,N) = (?M,?N) .
  op fv : Mset -> Set .
  ax fv(null) = emptyset .
  ax fv(M,N) = fv(M) U fv(N) .

  var P : Atom .
  vars A B : Prop0 .
  var T : Term .
  var X : Var .

  *** Identity
  rl      empty
     => --- ----
        ⊢ P,P^⊥ .

  *** Cut
  rl      (⊢ M,A) (⊢ N,A^⊥)
     => --- -------------
           ⊢ M,N .
```

[8] The multiset structure is one particular way of building in certain *structural rules*, in this case *exchange*. Many other such data structuring mechanisms are as well possible to build in, or to drop, desired structural properties.

```
*** Tensor
rl     (⊢ M,A) (⊢ B,N)
    => --- -----------
       ⊢ M,A ⊗ B,N .

*** Par
rl      ⊢ M,A,B
    => --- -------
       ⊢ M,A ⅋ B .

*** Plus
rl       ⊢ M,A
    => --- -------
       ⊢ M,A ⊕ B .

*** With
rl     (⊢ M,A) (⊢ M,B)
    => --- ------------
          ⊢ M,A & B .

*** Weakening
rl       ⊢ M
    => --- ----
       ⊢ M,?A .

*** Contraction
rl     ⊢ M,?A,?A
    => --- ------
          ⊢ M,?A .

*** Dereliction
rl     ⊢ M,A
    => --- ----
       ⊢ M,?A .

*** Storage
rl      ⊢ ?M,A
    => --- -----
       ⊢ ?M,!A .

*** Bottom
rl       ⊢ M
    => --- ---
       ⊢ M,⊥ .

*** One
rl     empty
    => --- --
       ⊢ 1 .

*** Top
rl       empty
    => --- ----
          ⊢ M,⊤ .

*** Universal
crl      ⊢ M,A
    => --- -----
       ⊢ M,∀X.A
    if  not(X is-in fv(M)) .

*** Existential
rl     ⊢ M,A[T/X]
    => --- ------
       ⊢ M.∃X.A .

endm
```

Given a linear theory $T = (C, S)$ (where we can assume that all the sequents in S are of the form $\vdash A_1, \ldots, A_n$), we instantiate the parameterized module `LL-SEQUENT[X]` using a functional module `C` that interprets the propositional constants in C, as in Section 4, and then extend it by adding a rule

```
rl empty => ⊢ A1,...,An .
```

for each sequent $\vdash A_1, \ldots, A_n$ in S, obtaining in this way a rewrite theory `LL-SEQUENT(T)`.

With this map we have also an immediate conservativity result:

Theorem 6.1 *Given a linear theory T, a linear logic sequent $\vdash A_1, \ldots, A_n$ is provable in linear logic from the axioms in T if and only if the sequent*

```
empty ⟶ ⊢ A1,...,An
```

is provable in rewriting logic from the rewrite theory `LL-SEQUENT(T)`.

It is very important to realize that the technique used in this conservative map of entailment systems is very general and it is in no way restricted to linear logic. Indeed, it can be applied to any sequent calculus, be it for intuitionistic, classical or any other logic. Moreover, it can even be applied to systems more general than traditional sequent calculi. Thus, a "sequent" can for example be a sequent presentation of natural deduction, a term assignment system, or even any predicate defined by structural induction in some way such that the proof is a kind of tree, as for example the operational semantics of CCS given later in Section 8 and any other use of the so-called structural operational semantics (see [53,27]). The general idea is to map a rule in the "sequent" system to a rewrite rule over a "configuration" of sequents or predicates, in such a way that the rewriting relation corresponds to provability of such a predicate.

7 Reflection

We give here a brief summary of the notion of a universal theory in a logic and of a reflective entailment system introduced in [9]. These notions axiomatize reflective logics within the theory of general logics [39]. We focus here on the simplest case, namely entailment systems. However, reflection at the proof calculus level—where not only sentences, but also proofs are reflected—is also very useful; the adequate definitions for that case are also in [9].

A reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. Two obvious metatheoretic notions that can be so reflected are theories and the entailment relation $\vdash$. This leads us to the notion of a universal theory. However, universality may not be absolute, but only relative to a class $\mathcal{C}$ of *representable* theories. Typically, for a theory to be representable at the object level, it must have a finitary description in some way—say, being recursively enumerable—so that it can be represented as a piece of language.

Given an entailment system $\mathcal{E}$ and a set of theories $\mathcal{C}$, a theory U is $\mathcal{C}$-*universal* if there is a function, called a *representation function*,

$$\overline{(_ \vdash _)} : \bigcup_{T \in \mathcal{C}} \{T\} \times sen(T) \longrightarrow sen(U)$$

such that for each $T \in \mathcal{C}$, $\varphi \in sen(T)$,

$$T \vdash \varphi \iff U \vdash \overline{T \vdash \varphi}.$$

If, in addition, $U \in \mathcal{C}$, then the entailment system $\mathcal{E}$ is called $\mathcal{C}$-*reflective*.

Note that in a reflective entailment system, since U itself is representable, representation can be iterated, so that we immediately have a "reflective tower"

$$T \vdash \varphi \iff U \vdash \overline{T \vdash \varphi} \iff U \vdash \overline{U \vdash \overline{T \vdash \varphi}} \ \ldots$$

Clavel and Meseguer have shown in [9,10] that indeed rewriting logic is reflective. That is, there is a rewrite theory $\mathcal{U}$ with a finite number of operations and rules that can simulate any other finitely presentable rewrite theory $\mathcal{R}$ in

the following sense: given any two terms t, t' in $\mathcal{R}$, there are corresponding terms $\langle \overline{\mathcal{R}}, \overline{t} \rangle$ and $\langle \overline{\mathcal{R}}, \overline{t'} \rangle$ in $\mathcal{U}$ such that we have

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

Moreover, it is often possible to reify inside rewriting logic itself a representation map $\mathcal{L} \to \mathit{OSRWLogic}$ for the finitely presentable theories of $\mathcal{L}$. Such a reification takes the form of a map between the abstract data types representing the finitary theories of $\mathcal{L}$ and of *OSRWLogic*. In this section we illustrate this powerful idea with the linear logic mapping defined in Section 4.

We have defined a linear theory T as a finite set C of propositional constants together with a finite set S of sequents of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$, where each A_i and B_j is a linear logic formula built from the constants in C. Note that with this definition, all linear theories are finitely presentable. First, we define an abstract data type `LL-ADT` to represent linear theories. A linear theory is represented as a term `<C | G>`, where `C` is a list of propositional constants (that is, identifiers), and `G` is a list of sequents written in the usual way. Moreover, all the propositional constants in `G` must be included in `C`. To enforce this condition, we use a sort constraint [44], which is introduced with the keyword `sct` and defines a subsort `LLTheory` of a sort `LLTheory?` by means of the given condition. In the functional module below, we do not give the equations defining the auxiliary functions `const` that extracts the constants of a list of sequents, and the list containment predicate `_=<_`. These functions are needed to write down the sort constraint for theories.

```
fmod LL-ADT is
  protecting QID .
  sorts Ids Formula Formulas Sequent .
  sorts Sequents LLTheory? LLTheory .

  subsort Id < Formula .
  ops 1 0 ⊥ ⊤ : -> Formula .
  op _⊗_ : Formula Formula -> Formula .
  op _⅋_ : Formula Formula -> Formula .
  op _⊕_ : Formula Formula -> Formula .
  op _&_ : Formula Formula -> Formula .
  op !_ : Formula -> Formula .
  op ?_ : Formula -> Formula .
  op _⊥ : Formula -> Formula .

  subsort Formula < Formulas .
  op null : -> Formulas .
  op _,_ : Formulas Formulas -> Formulas [assoc comm id: null] .

  op _⊢_ : Formulas Formulas -> Sequent .

  subsort Id < Ids .
  op nil : -> Ids .
```

```
  op _,_ : Ids Ids -> Ids [assoc id: nil] .

  subsort Sequent < Sequents .
  op nil : -> Sequents .
  op _,_ : Sequents Sequents -> Sequents [assoc id: nil] .

  op <_|_> : Ids Sequents -> LLTheory? .

  var C : Ids .
  var G : Sequents .
  sct <C | G> : LLTheory if const(G) =< C .

  eq ...
  *** several equations defining the auxiliary operations
  *** "const" and "_=<_" used in the sort constraint condition
  eq ...
endfm
```

An order-sorted rewrite theory has much more structure, and therefore the corresponding RWL-ADT is more complex, but the basic ideas are completely similar as we sketch here. First we have an order-sorted signature, declaring sorts, subsorts, constants, operations and variables. Then, in addition, we have equations and rules. Thus, a finitely presentable rewrite theory is represented as a term <S | E | R>, where S is a term representing a signature, E is a list of equations, and R is a list of rules. In turn, the term S has the form <T ; B ; C ; O ; V> where each subterm corresponds to a component of a signature as mentioned before. In addition, several sort constraints are necessary to ensure for example that the variables used in equations and rules are included in the list of variables. Just to give the flavor of the construction, here is a small fragment of the module RWL-ADT, where we have omitted most of the list constructors, operations to handle conditional equations and rules, and sort constraints.

```
sorts Sort Subsort Constant Op Var .
sorts Term Equation Rule Signature RWLTheory .

op sort{_} : Id -> Sort .
subsort Sort < Sorts .
op nil : -> Sorts .
op __ : Sorts Sorts -> Sorts [assoc id: nil] .

op (_<_) : Id Id -> Subsort .
subsort Subsort < Subsorts .

op (cons{_}:sort{_}) : Id Id -> Constant .
subsort Constant < Constants .
op nil : -> Constants .
op _,_ : Constants Constants -> Sorts [assoc id: nil] .
```

```
op (op{_}:_->sort{_}) : Id Sorts Id -> Op .
subsort Op < Ops .

op (var{_}:sort{_}) : Id Id -> Var .
subsort Var < Vars .

op <_;_;_;_;_> : Sorts Subsorts Constants Ops Vars
                                              -> Signature .
subsort Var < Term .
subsort Constant < Term .
subsort Term < Terms .
op nil : -> Terms .
op op{_}[_] : Id Terms -> Term .
op _,_ : Terms Terms -> Terms [assoc id: nil] .

op (_=_) : Term Term -> Equation .
subsort Equation < Equations .

op (_=>_) : Term Term ->  Rule .
subsort Rule < Rules .

op <_|_|_> : Signature Equations Rules -> RWLTheory .
```

In this way, the rewrite theory LINLOG presented in Section 4 gives rise to a term in RWLTheory that we denote

$$\texttt{<<}T_{LL}\ \texttt{;}\ B_{LL}\ \texttt{;}\ C_{LL}\ \texttt{;}\ O_{LL}\ \texttt{;}\ V_{LL}\texttt{>}\ \texttt{|}\ E_{LL}\ \texttt{|}\ R_{LL}\texttt{>}.$$

Having defined the abstract data types to represent both linear and rewrite theories, we define a function $\overline{\Phi}$ mapping a term in LLTheory representing a linear theory T to a term in RWLTheory representing the corresponding rewrite theory LINLOG(T) as defined in Section 4.

The representation <C | F1 $\vdash$ G1,...,Fn $\vdash$ Gn> is mapped by $\overline{\Phi}$ to the following term

$$\texttt{<<}T_{LL}\ \texttt{;}\ B_{LL}\ \texttt{;}\ \texttt{cons(C),}C_{LL}\ \texttt{;}\ O_{LL}\ \texttt{;}\ V_{LL}\texttt{>}\ \texttt{|}\ E_{LL}\ \texttt{|}$$
$$R_{LL}\texttt{,([tensor(F1)] => [par(G1)]),...,([tensor(Fn)] => [par(Gn)])>}$$

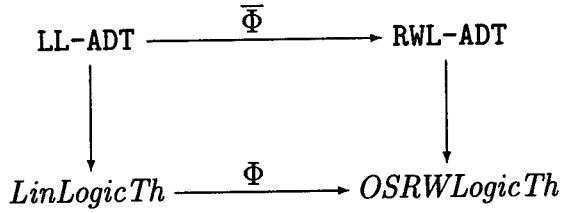
where the auxiliary operations cons, tensor and par are defined as follows, and correspond exactly to the description in Section 4.

```
op tensor : Formulas -> Formula .
op par : Formulas -> Formula .
op cons : Ids -> Constants .

vars F1 F2 : Formulas .
var I : Id .
var L : Ids .
```

```
eq tensor(null) = 1 .
eq tensor(F1,F2) = tensor(F1) ⊗ tensor(F2) .
eq par(null) = ⊥ .
eq par(F1,F2) = par(F1) ⅋ par(F2) .
eq cons(nil) = nil .
eq cons(I,L) = (cons{I}:sort{Atom}),cons(L) .
```

We can summarize the reification $\overline{\Phi}$: LL-ADT $\longrightarrow$ RWL-ADT of the map of logics Φ : *LinLogic* $\longrightarrow$ *OSRWLogic* we have just defined by means of the following commutative diagram:



This method is completely general, in that it should apply to any effectively presented map of logics $\Psi : \mathcal{L} \longrightarrow$ *RWLogic* that maps finitely presentable theories in $\mathcal{L}$ to finitely presentable theories in rewriting logic. Indeed, the effectiveness of Ψ should exactly mean that the corresponding $\overline{\Psi}$: $\mathcal{L}$-ADT $\longrightarrow$ RWL-ADT is a computable function and therefore, by the metatheorem of Bergstra and Tucker [4], that it is specifiable by a finite set of Church-Rosser and terminating equations inside rewriting logic.

8 CCS

Milner's *Calculus of Communicating Systems* (CCS) [47,48] is among the best well-known and studied concurrency models, and has become the paradigmatic example of an entire approach to "process algebras." We give a very brief introduction to CCS, referring the reader to Milner's book [48] for motivation and a comprehensive treatment; then, we give a formulation of CCS in rewriting logic and show its conservativity.

We assume a set A of *names*; the elements of the set $\overline{A} = \{\overline{a} \mid a \in A\}$ are called *co-names*, and the members of the (disjoint) union $\mathcal{L} = A \cup \overline{A}$ are *labels* naming ordinary actions. The function $a \mapsto \overline{a}$ is extended to $\mathcal{L}$ by defining $\overline{\overline{a}} = a$. There is a special action called *silent action* and denoted τ, intended to represent internal behaviour of a system, and in particular the synchronization of two processes by means of actions a and $\overline{a}$. Then the set of *actions* is $\mathcal{L} \cup \{\tau\}$. The set of processes is intuitively defined as follows:

- 0 is an inactive process that does nothing.
- If α is an action and P is a process, $\alpha.P$ is the process that performs α and subsequently behaves as P.
- If P and Q are processes, $P + Q$ is the process that may behave as either P or Q.
- If P and Q are processes, $P|Q$ represents P and Q running concurrently

with possible communication via synchronization of the pair of ordinary actions a and $\overline{a}$.

- If P is a process and $f : \mathcal{L} \to \mathcal{L}$ is a relabelling function such that $f(\overline{a}) = \overline{f(a)}$, $P[f]$ is the process that behaves as P but with the actions relabelled according to f, assuming $f(\tau) = \tau$.
- If P is a process and $L \subseteq \mathcal{L}$ is a set of ordinary actions, $P\backslash L$ is the process that behaves as P but with the actions in $L \cup \overline{L}$ prohibited.
- If P is a process, I is a process identifier, and $I =_{def} P$ is a defining equation where P may recursively involve I, then I is a process that behaves as P.

This intuitive explanation can be made precise in terms of the following structural operational semantics that defines a labelled transition system for CCS processes.

Action:

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}$$

Summation:

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

Composition:

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

Relabelling:

$$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

Restriction:

$$\frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L} \quad \alpha \notin L \cup \overline{L}$$

Definition:

$$\frac{P \xrightarrow{\alpha} P'}{I \xrightarrow{\alpha} P'} \quad I =_{def} P$$

We now show how CCS can be described and given semantics in rewriting logic.

```
fth LABEL is
  sort Label . *** ordinary actions
  op ~_ : Label -> Label .
  var N : Label .
  eq ~~N = N .
endft
```

```
fmod ACTION[X :: LABEL] is
  sort Act .
  subsort Label < Act .
  op tau : -> Act .  *** silent action
endfm

fth PROCESSID is
  sort ProcessId . *** process identifiers
endft

fmod PROCESS[X :: LABEL, Y :: PROCESSID] is
  protecting ACTION[X] .
  sort Process .
  subsort ProcessId < Process .
  op 0 : -> Process .                          *** inaction
  op _._ : Act Process -> Process .            *** prefix
  op _+_ : Process Process -> Process [assoc comm idem id: 0] .
                                               *** summation
  op _|_ : Process Process -> Process [assoc comm id: 0] .
                                               *** composition
  op _[_/_] : Process Label Label -> Process .
                    *** relabelling: [b/a] relabels "a" to "b"
  op _\_ : Process Label -> Process .  *** restriction
endfm
```

Before defining the operational semantics of CCS processes, we need an auxiliary module in order to build contexts in which process identifiers can be associated with processes, providing in this way recursive definitions of processes. A sort constraint [44], which is introduced with the keyword `sct` and defines a subsort `Context` by means of a condition, is used to enforce the requirement that the same process identifier cannot be associated with two different processes in a context.

```
fmod CCS-CONTEXT[X :: LABEL, Y :: PROCESSID] is
  protecting PROCESS[X,Y] .
  sorts Def Context .
  op (_ =def _) : ProcessId Process -> Def .
  protecting LIST[ProcessId]*(op _;_ to __) .
  protecting LIST[Def]*(sort List to Context?) .
  subsorts Def < Context < Context? .
  op nil : -> Context .
  op pid : Context? -> List .
  var X : ProcessId .
  var P : Process .
  var C : Context .
  vars D D' : Context? .
  eq pid(nil) = nil .
```

```
  eq pid((X =def P)) = X .
  eq pid(D;D') = pid(D) pid(D') .
  sct (X =def P);C : Context if not(X in pid(C)) .
endfm
```

The semantics of CCS processes is usually defined relative to a given context that provides defining equations for all the necessary process identifiers [48, Section 2.4]. The previous module defines the data type of all contexts. We now need to parameterize the module defining the CCS semantics by the choice of a context. This is accomplished by means of the following theory that picks up a context in the sort `Context`.

```
fth CCS-CONTEXT*[X :: LABEL, Y :: PROCESSID] is
  protecting CCS-CONTEXT[X,Y] .
  op context : -> Context .
endft
```

As in the case of linear logic, we have two possibilities in order to write the operational semantics for CCS by means of rewrite rules. On the one hand, we can interpret a transition $P \xrightarrow{\alpha} P'$ as a rewrite, so that the above operational semantics rules become conditional rewrite rules. On the other hand, the transition $P \xrightarrow{\alpha} P'$ can be seen as a term, forming part of a configuration, in such a way that the semantics rules correspond to rewrite rules, as a particular case of the general mapping of sequent systems into rewriting logic that we have presented in Section 6. Here we describe the first possibility and refer the reader to [36, Section 5.3] for the second one.

```
mod CCS1[X :: LABEL, Y :: PROCESSID, C :: CCS-CONTEXT*[X,Y]] is
  sort ActProcess .
  subsort Process < ActProcess .
  op {_}_ : Act ActProcess -> ActProcess .
  *** {A}P means that the process P has performed the action A
  vars L M : Label .
  var A : Act .
  vars P P' Q Q' : Process .
  var X : ProcessId .
  *** Prefix
  rl A . P => {A}P .

  *** Summation
  crl P + Q => {A}P'  if  P => {A}P' .

  *** Composition
  crl P | Q => {A}(P' | Q)  if  P => {A}P' .
  crl P | Q => {tau}(P' | Q')  if  P => {L}P' and Q => {~L}Q' .

  *** Restriction
  crl P \ L => {A}(P' \ L)  if  P => {A}P' and
                                not(A == L) and not(A == ~L) .
```

```
  *** Relabelling
  crl P[M / L] => {M}(P'[M / L])  if  P => {L}P' .
  crl P[M / L] => {~M}(P'[M / L])  if  P => {~L}P' .
  crl P[M / L] => {A}(P'[M / L])  if  P => {A}P' and
                                  not(A == L) and not(A == ~L) .
  *** Definition
  crl X => {A}P'  if  (X =def P) in context and P => {A}P' .
endm
```

In the above module, the rewrite rules have the property of being sort-increasing, i.e., in a rule $[t] \longrightarrow [t']$ the least sort of $[t']$ is bigger than the least sort of $[t]$. Thus, one rule cannot be applied unless the resulting term is well-formed. This prevents, for example, rewrites of the following form:

$$\texttt{\{A\}(P | Q)} \longrightarrow \texttt{\{A\}(\{B\}P' | \{C\}Q')}$$

because the term on the righthand side is not well formed according to the order-sorted signature of the module `CCS1[X,Y,C[X,Y]]`. More precisely, the *Congruence* rule of order-sorted rewriting logic, like the corresponding rule of order-sorted algebra [22], cannot be applied unless the resulting terms $f(t_1, \ldots, t_n)$ are well formed according to the given order-sorted signature. To illustrate this point further, although $\texttt{A.P} \longrightarrow \texttt{\{A\}P}$ is a correct instance of the `Prefix` rewrite rule, we cannot use the *Congruence* rule to derive

$$\texttt{(A.P) | Q} \longrightarrow \texttt{(\{A\}P) | Q}$$

because the second term `({A}P) | Q` is not well formed.

The net effect of this restriction is that an `ActProcess` term of the form `{A1}...{Ak}P` can only be rewritten into another term of the same form `{A1}...{Ak}{B}P'`, being $\texttt{P} \longrightarrow \texttt{\{B\}P'}$ a `CCS1[X,Y,C[X,Y]]`-rewrite. As another example, a process of the form `A.B.P` can be rewritten first into `{A}B.P` and then into `{A}{B}P`, but cannot be rewritten into `A.{B}P`, because this last term is not well formed. After this discussion, it is easy to see that we have the following conservativity result.

Theorem 8.1 *Given a CCS process P, there are processes $P_1, \ldots, P_{k-1}$ such that*

$$P \xrightarrow{a_1} P_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{k-1}} P_{k-1} \xrightarrow{a_k} P'$$

if and only if `P` *can be rewritten into* `{a1}...{ak}P'` *using the rules in the module* `CCS1[X,Y,C[X,Y]]`.

Note also that, since the operators `_+_` and `_|_` are declared commutative, one rule is enough for each one, instead of the two rules in the original presentation. On the other hand, we need three rules for relabelling, due to the representation of the relabelling function.

9 Constraint solving

Deduction can in many cases be made much more efficient by making use of *constraints* that can drastically reduce the search space, and for which

special purpose constraint solving algorithms can be much faster than the alternative of expressing everything in a unique deduction mechanism such as some form of resolution. Typically, constraints are symbolic expressions associated with a particular *theory*, and a constraint solving algorithm uses intimate knowledge about the truths of the theory in question to find solutions for those expressions by transforming them into expressions in *solved form*.

One of the simplest examples is provided by standard syntactic unification, the constraint solver for resolution in first-order logic without equality and in particular for Prolog, where the constraints in question are equalities between terms in a free algebra, i.e., in the so-called Herbrand universe. There are however many other constraints and constraint solving algorithms that can be used to advantage in order to make the representation of problems more expressive and logical deduction more efficient. For example,

- *Semantic unification* (see for example [30]), which corresponds to solving equations in a given equational theory.
- *Sorted unification*, either many-sorted or order-sorted [58,54,45,30], where type constraints are added to variables in equations.
- *Higher-order unification* [28], which corresponds to solving equations between λ-expressions.
- *Disunification* [13], which corresponds to solving not only equalities but also negated equalities.
- *Solution of equalities and inequalities in a numerical theory*, as for example the solution of numerical constraints built into the constraint logic programming language $CLP(\mathcal{R})$ [29] and in other languages.

A remarkable property shared by most constraint-solving processes, and already implicit in the approach to syntactic unification problems proposed by Martelli and Montanari [34], is that the process of solving constraints can be naturally understood as one of applying transformations to a set or multiset of constraints. Furthermore, many authors have realized that the most elegant and simple way to specify, prove correct, or even implement many constraint solving problems is by expressing those transformations as rewrite rules (see for example [21,30,12,13]). In particular, the survey by Jouannaud and Kirchner [30] makes this viewpoint the cornerstone of a unified conceptual approach to unification.

For example, the so-called *decomposition* transformation present in syntactic unification and in a number of other unification algorithms can be expressed by a rewrite rule of the form

```
f(t1,...,tn) =?= f(t'1,...,t'n)
        => (t1 =?= t'1) ...  (tn =?= t'n)
```

where in the righthand side multiset union has been expressed by juxtaposition.

Although the operational semantics of such rewrite rules is very obvious and intuitive, their logical or mathematical semantics has remained ambiguous. Although appeal is sometimes made to equational logic as the framework

in which such rules exist, the fact that many of these rules are nondeterministic, so that, except for a few exceptions such as syntactic unification, there is in general not a unique solution but rather a, sometimes infinite, set of solutions, makes an interpretation of the rewrite rules as equations highly implausible and potentially contradictory.

We would like to suggest that rewriting logic provides a very natural framework in which to interpret rewrite rules of this nature and, more generally, deduction processes that are nondeterministic in nature and involve the exploration of an entire space of solutions. Since in rewriting logic rewrite rules go only in one direction and its models do not assume either the identification of the two sides of a rewrite step, or even the possible reversal of such a step, all the difficulties involved in an equational interpretation disappear.

Such a proposed use of rewriting logic for constraint solving and constraint programming seems very much in the spirit of recent rewrite rule approaches to constrained deduction such as those of C. Kirchner, H. Kirchner, and M. Rusinovitch [31], Bachmair, Ganzinger, Lynch, and Snyder [2], and Nieuwenhuis and Rubio [50]. In particular, the ELAN language of C. Kirchner, H. Kirchner, and M. Vittek [32,5] proposes an approach to the prototyping of constraint solving languages similar in some ways to the one that would be natural using a Maude interpreter.

10 Concluding remarks

Rewriting logic has been proposed as a logical framework that seems particularly promising for representing logics, and its use for this purpose has been illustrated in detail by a number of examples. The general way in which such representations are achieved is by:

- Representing formulas or, more generally, proof-theoretic structures such as sequents, as *terms* in an order-sorted equational data type whose equations express structural axioms natural to the logic in question.
- Representing the rules of deduction of a logic as rewrite rules that transform certain patterns of formulas into other patterns modulo the given structural axioms.

Besides, the theory of general logics [39] has been used as both a method and a criterion of adequacy for defining these representations as conservative maps of logics or of entailment systems. From this point of view, our tentative conclusion is that, at the level of entailment systems, rewriting logic should in fact be able to represent any finitely presented logic via a conservative map, for any reasonable notion of "finitely presented logic." Making this tentative conclusion definite will require proposing an intuitively reasonable formal version of such a notion in a way similar to previous proposals of this kind by Smullyan [56] and Feferman [15].

In some cases, such as for equational logic, Horn logic with equality, and linear logic, we have in fact been able to represent logics in a much stronger sense, namely by conservative maps of logics that also map the models [36]. Of

course, such maps are much more informative, and may afford easier proofs, for example for conservativity. However, one should not expect to find representations of this kind for logics whose model theory is very different from that of rewriting logic.

We have also shown how the fact that rewriting logic is reflective greatly enhances its capabilities as a logical framework, by allowing the metalevel representation maps $\mathcal{L} \longrightarrow \mathit{RWLogic}$ to be reified inside rewriting logic itself.

The uses of rewriting logic as a semantic framework for the specification of languages, systems, and models of computation have also been discussed and illustrated with examples. Such uses include the specification and prototyping of concurrent models of computation and concurrent object-oriented systems, of general programming languages, of automated deduction systems and logic programming languages that use constraints, and of logical representation of action and change in AI [36,42].

From a pragmatic point of view, the main goal of this study is to serve as a guide for the design and implementation of a theoretically-based high-level system in which it can be easy to define logics and to perform deductions in them, and in which a very wide variety of systems, languages, and models of computation can similarly be specified and prototyped. Having this goal in mind, the following features seem particularly useful:

- *Executability*, which is not only very useful for prototyping purposes, but is in practice a must for debugging specifications of any realistic size.
- *User-definable abstract syntax*, which can be specified as an order-sorted equational data type with the desired structural axioms.
- *Modularity and parameterization* [9], which can make specifications very readable and reusable by decomposing them in small understandable pieces that are as general as possible.
- *Simple and general logical semantics*, which can naturally express both logical deductions and concurrent computations.

These features are supported by the Maude interpreter [8]. A very important additional feature that the Maude interpreter has is good support for flexible and expressive strategies of evaluation [8,10], so that the user can explore the space of rewritings in intelligent ways.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, Explicit substitutions, *Journal of Functional Programming* **1**(4) (1991) 375–416.

[2] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder, Basic paramodulation and superposition, in: D. Kapur, ed., *Proc. 11th. Int. Conf. on Automated Deduction, Saratoga Springs, NY* (LNAI 607, Springer-Verlag, 1992) 462–476.

[9] Parameterization is based on the existence of relatively free algebras in rewriting logic, which generalizes the existence of initial algebras.

[3] D. A. Basin and R. L. Constable, Metalogical frameworks, in: G. Huet and G. Plotkin, eds., *Logical Environments* (Cambridge University Press, 1993) 1–29.

[4] J. Bergstra and J. Tucker, Characterization of computable data types by means of a finite equational specification method, in: J. W. de Bakker and J. van Leeuwen, eds., *Proc. 7th. Int. Colloquium on Automata, Languages and Programming* (LNCS 81, Springer-Verlag, 1980) 76–90.

[5] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek, ELAN: A logical framework based on computational systems, in: J. Meseguer, ed., *Proc. First Int. Workshop on Rewriting Logic and its Applications* (ENTCS 4, Elsevier, 1996) [This volume].

[6] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer, Specification and proof in membership equational logic, paper in preparation (1996).

[7] M. Cerioli and J. Meseguer, May I borrow your logic? (Transporting logical structure along maps), to appear in Theoretical Computer Science (1996).

[8] M. G. Clavel, S. Eker, P. Lincoln, and J. Meseguer, Principles of Maude, in: J. Meseguer, ed., *Proc. First Int. Workshop on Rewriting Logic and its Applications* (ENTCS 4, Elsevier, 1996) [This volume].

[9] M. G. Clavel and J. Meseguer, Axiomatizing reflective logics and languages, in: G. Kiczales, ed., *Proc. Reflection'96* (San Francisco, USA, April 1996) 263–288.

[10] M. G. Clavel and J. Meseguer, Reflection in rewriting logic, in: J. Meseguer, ed., *Proc. First Int. Workshop on Rewriting Logic and its Applications* (ENTCS 4, Elsevier, 1996) [This volume].

[11] J. R. B. Cockett and R. A. G. Seely, Weakly distributive categories, in: M. P. Fourman, P. T. Johnstone, and A. M. Pitts, eds., *Applications of Categories in Computer Science* (Cambridge University Press, 1992) 45–65.

[12] H. Comon, Equational formulas in order-sorted algebras, in: M. S. Paterson, ed., *Proc. 17th. Int. Colloquium on Automata, Languages and Programming, Warwick, England* (LNCS 443, Springer-Verlag, 1990) 674–688.

[13] H. Comon, Disunification: A survey, in: J.-L. Lassez and G. Plotkin (eds.), *Computational Logic: Essays in Honor of Alan Robinson* (The MIT Press, 1991) 322–359.

[14] N. Dershowitz and J.-P. Jouannaud, Rewrite systems, in: J. van Leeuwen *et al.*, eds., *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics* (The MIT Press/Elsevier, 1990) 243–320.

[15] S. Feferman, Finitary inductively presented logics, in: R. Ferro *et al.*, eds., *Logic Colloquium'88* (North-Holland, 1989) 191–220.

[16] A. Felty and D. Miller, Encoding a dependent-type λ-calculus in a logic programming language, in: M. E. Stickel, ed., *Proc. 10th. Int. Conf. on Automated Deduction, Kaiserslautern, Germany* (LNAI 449, Springer-Verlag, 1990) 221–235.

[17] P. Gardner, *Representing Logics in Type Theory*, Ph.D. Thesis, Department of Computer Science, University of Edinburgh (1992).

[18] J.-Y. Girard, Linear logic, *Theoretical Computer Science* **50** (1987) 1–102.

[19] J. A. Goguen and R. M. Burstall, Introducing institutions, in: E. Clarke and D. Kozen, eds., *Proc. Logics of Programming Workshop* (LNCS 164, Springer-Verlag, 1984) 221–256.

[20] J. A. Goguen and R. M. Burstall, Institutions: Abstract model theory for specification and programming, *Journal of the Association for Computing Machinery* **39**(1) (1992) 95–146.

[21] J. A. Goguen and J. Meseguer, Software for the rewrite rule machine, in: *Proc. of the Int. Conf. on Fifth Generation Computer Systems* (ICOT, Tokyo, Japan, 1988) 628–637.

[22] J. A. Goguen and J. Meseguer, Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions, and partial operations, *Theoretical Computer Science* **105** (1992) 217–273.

[23] J. A. Goguen, A. Stevens, K. Hobley, and H. Hilberdink, 2OBJ: A meta-logical framework based on equational logic, *Philosophical Transactions of the Royal Society, Series A* **339** (1992) 69–86.

[24] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, Introducing OBJ, Technical report SRI-CSL-92-03, Computer Science Laboratory, SRI International (March 1992). To appear in J. A. Goguen and G. Malcolm, eds., *Software Engineering with OBJ: Algebraic Specification in Practice* (Cambridge University Press).

[25] R. Harper, F. Honsell, and G. Plotkin, A framework for defining logics, *Journal of the Association for Computing Machinery* **40**(1) (1993) 143–184.

[26] R. Harper, D. Sannella, and A. Tarlecki, Structure and representation in LF, in: *Proc. Fourth Annual IEEE Symp. on Logic in Computer Science* (Asilomar, California, June 1989) 226–237.

[27] M. Hennessy, *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics* (John Wiley and Sons, 1990).

[28] G. Huet, A unification algorithm for typed lambda calculus, *Theoretical Computer Science* **1**(1) (1973) 27–57.

[29] J. Jaffar and J. Lassez, Constraint logic programming, in: *Proc. 14th. ACM Symp. on Principles of Programming Languages* (Munich, Germany, 1987) 111-119.

[30] J.-P. Jouannaud and C. Kirchner, Solving equations in abstract algebras: A rule-based survey of unification, in: J.-L. Lassez and G. Plotkin, eds., *Computational Logic: Essays in Honor of Alan Robinson* (The MIT Press, 1991) 257–321.

[31] C. Kirchner, H. Kirchner, and M. Rusinowitch, Deduction with symbolic constraints, *Revue Francaise d'Intelligence Artificielle* **4**(3) (1990) 9–52.

[32] C. Kirchner, H. Kirchner, and M. Vittek, Designing constraint logic programming languages using computational systems, in: V. Saraswat and P. van Hentenryck, eds., *Principles and Practice of Constraint Systems: The Newport Papers* (The MIT Press, 1995) 133–160.

[33] C. Laneve and U. Montanari, Axiomatizing permutation equivalence in the λ-calculus, in: H. Kirchner and G. Levi, eds., *Proc. Third Int. Conf. on Algebraic and Logic Programming, Volterra, Italy* (LNCS 632, Springer-Verlag, 1992) 350–363.

[34] A. Martelli and U. Montanari, An efficient unification algorithm, *ACM Transactions on Programming Languages and Systems* **4**(2) (1982) 258-282.

[35] N. Martí-Oliet and J. Meseguer, From Petri nets to linear logic through categories: A survey, *International Journal of Foundations of Computer Science* **2**(4) (1991) 297–399.

[36] N. Martí-Oliet and J. Meseguer, Rewriting logic as a logical and semantic framework, Technical report SRI-CSL-93-05, Computer Science Laboratory, SRI International (August 1993). To appear in D. M. Gabbay, ed., *Handbook of Philosophical Logic* (Kluwer Academic Publishers).

[37] N. Martí-Oliet and J. Meseguer, General logics and logical frameworks, in: D. Gabbay, ed., *What Is a Logical System?* (Oxford University Press, 1994) 355–392.

[38] S. Matthews, A. Smaill, and D. Basin, Experience with FS_0 as a framework theory, in: G. Huet and G. Plotkin, eds., *Logical Environments* (Cambridge University Press, 1993) 61–82.

[39] J. Meseguer, General logics, in: H.-D. Ebbinghaus *et al.*, eds., *Logic Colloquium'87* (North-Holland, 1989) 275–329.

[40] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, *Theoretical Computer Science* **96** (1992) 73–155.

[41] J. Meseguer, A logical theory of concurrent objects and its realization in the Maude language, in: G. Agha, P. Wegner, and A. Yonezawa, eds., *Research Directions in Object-Based Concurrency* (The MIT Press, 1993) 314–390.

[42] J. Meseguer, Rewriting logic as a semantic framework for concurrency: A progress report, in: *Proc. CONCUR'96* (LNCS, Springer-Verlag, 1996).

[43] J. Meseguer, Membership algebra, Lecture at the *Dagstuhl Seminar on Specification and Semantics* (July 1996). Extended version in preparation.

[44] J. Meseguer and J. A. Goguen, Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems, *Information and Computation* **104** (1993) 114–158.

[45] J. Meseguer, J. A. Goguen, and G. Smolka, Order-sorted unification, *Journal of Symbolic Computation* **8** (1989) 383–413.

[46] J. Meseguer and T. Winkler, Parallel programming in Maude, in: J. P. Banâtre and D. Le Métayer, eds., *Research Directions in High-Level Parallel Programming Languages* (LNCS 574, Springer-Verlag, 1992) 253–293.

[47] R. Milner, *A Calculus of Communicating Systems* (LNCS 92, Springer-Verlag, 1980).

[48] R. Milner, *Communication and Concurrency* (Prentice Hall, 1989).

[49] G. Nadathur and D. Miller, An overview of λProlog, in: K. Bowen and R. Kowalski, eds., *Fifth Int. Joint Conf. and Symp. on Logic Programming* (The MIT Press, 1988) 810–827.

[50] R. Nieuwenhuis and A. Rubio, Theorem proving with ordering constrained clauses, in: D. Kapur, ed., *Proc. 11th. Int. Conf. on Automated Deduction, Saratoga Springs, NY* (LNAI 607, Springer-Verlag, 1992) 477–491.

[51] L. Paulson, The foundation of a generic theorem prover, *Journal of Automated Reasoning* **5** (1989) 363–397.

[52] F. Pfenning, Elf: A language for logic definition and verified metaprogramming, in: *Proc. Fourth Annual IEEE Symp. on Logic in Computer Science* (Asilomar, California, June 1989) 313–322.

[53] G. D. Plotkin, A structural approach to operational semantics, Technical report DAIMI FN-19, Computer Science Department, Aarhus University (September 1981).

[54] M. Schmidt-Schauss, *Computational Aspects of Order-Sorted Logic with Term Declarations* (LNCS 395, Springer-Verlag, 1989).

[55] R. A. G. Seely, Linear logic, $*$-autonomous categories and cofree coalgebras, in: J. W. Gray and A. Scedrov, eds., *Categories in Computer Science and Logic, Boulder, June 1987* (Contemporary Mathematics 92, AMS, 1989) 371–382.

[56] R. M. Smullyan, *Theory of Formal Systems* (Princeton University Press, 1961).

[57] A. S. Troelstra, *Lectures on Linear Logic* (CSLI, Stanford University, 1992).

[58] C. Walther, A classification of many-sorted unification theories, in: J. H. Siekmann, ed., *Proc. 8th. Int. Conf. on Automated Deduction, Oxford, England* (LNCS 230, Springer-Verlag, 1986) 525–537.

Electronic Notes in Theoretical Computer Science 4 (1996)

Foundations of Behavioural Specification in Rewriting Logic

Răzvan Diaconescu [1]

Graduate School of Information Science
Japan Advanced Institute of Science and Technology
Ishikawa, Japan

Abstract

We extend behavioural specification based on hidden sorts to rewriting logic by constructing a hybrid between the two underlying logics. This is achieved by defining a concept of behavioural satisfaction for rewriting logic. Our approach is semantic in that it is based on a general construction on models, called *behaviour image*, which uses final models in an essential way. However we provide syntactic characterisations for the for the behavioural satisfaction relation, thus opening the door for shifting recent advanced proof techniques for behavioural satisfaction to rewriting logic. We also show that the rewriting logic behavioural satisfaction obeys the so-called "satisfaction condition" of the theory of institutions, thus providing support for OBJ style modularisation for this new paradigm.

1 Introduction

This research aims at integrating two different semantic approaches on objects and concurrency by internalising behavioural specification [12] to *[conditional] rewriting logic* (abbreviatted **RWL**) [18]. We provide with a logic underlying this integration of the two specification paradigms, which we call **hidden sorted rewriting logic** (abreviatted **HSRWL**), and which is a hybrid between RWL and Goguen's *hidden sorted algebra* (abreviatted **HSA**), i.e., the logic underlying behavioural specification in equational logic. Both HSA and RWL have their origin within the rich tradition of algebraic specification, however their approach to objects and concurrency is quite different. HSRWL aims at providing a unitary framework encoding all important aspects of concurrent objects by combining the HSA approach to behavioural specification of objects (featuring local states, attributes, methods, classes, inheritance, etc.) with the RWL approach to concurrent distributed systems. Thus HSRWL unifies the two constituent sub-logics within a single but flexible framework.

[1] On leave from the Institute of Mathematics of the Romanian Academy.

© *1996 Elsevier Science B. V.*

This is hardly a new idea, the semantics of many multi-paradigm systems where treated similarly, i.e., by unifying the logics involved.

HSRWL is based on a definition of behavioural satisfaction for RWL. We approach behavioural satisfaction for RWL from a semantic angle (as opposed to the syntactic one of [12,14]) by following the general methodology introduced in [3] but adapted to the rather complex case of RWL. This approach highlights a construction on models, called *behaviour image*, using final models; this significantly generalises the concept of "behavioural equivalence" central to the HSA theory. Behaviour images do more than equating [elements but also transitions in the RWL case] under behavioural equivalence, they also add new "behaviour transitions". So we argue that the concept of behaviour image is more fundamental than behaviour equivalence. Another advantage of this semantic approach, also very transparent in [3], is that it is actually independent of sentences, so it can be directly used for many kinds of sentences without regard of their complexity.

As mentioned above, final models play a crucial rôle in our definition of behavioural satisfaction for RWL. We show that final models can be obtained naturally as a (proper) Kan extension, which provides a compact formula for the final models.

We also study a more refined development of behavioural satisfaction for RWL which takes into account the possible transitions between the observations on the states of the system (which are naturally induced by the trasitions at the level of data). This amounts in principle to a *2-version* (we borrow the terminology from 2-categories which play an important technical rôle here) of our theory, and the development of the concepts and results for both the simple and the 2-version is done in parallel. At the end we analyse the relationship between behavioural satisfaction and 2-behavioural satisfaction and provide some sufficient conditions for them to coincide. For example, they coincide whenever there is at most one transition between the elements of data, which in most aplications is a desirable condition.

Our work concentrates at providing the semantic foundations for behavioural specification in RWL, which can be used as a solid logical basis for proving behavioural properties of concurrent distributed systems. This requires a proof theory for the behavioural satisfaction which is actually essential in applications. We give syntactic characterisations for behavioural satisfaction analogous to the original HSA definition of behavioural satisfaction using contexts. This opens the door for using the advanced techniques developed for HSA, such as the so-called "coinduction" of [14].

Behavioural specification supports the powerful module system á la OBJ [15] featuring parameterised programming and module expressions. This is based on the HSRWL institution, thus enabling the direct application of the semantics of module systems developed using institutions [11,8]. The HSRWL institution depends on some conditions on the signature morphisms which were first discovered by Goguen in [10], the most important naturally corresponding to the encapsulation of classes and sub-classes from object-oriented programming. HSRWL adds a new condition corresponding to "encapsulation

of structural axioms", this is due to computation modulo structural axioms being treated as a first-class citizen in the definition of RWL (see [18]).

Finally, we hope HSRWL can be used as a framework for the semantics of systems effectively combining behavioural specification and RWL. Such an example is CafeOBJ [9] (whose semantics provide sin fact the main motivation for this work), and in the Appendix we provide a small example written in CafeOBJ whose main purpose is to illustrate in detail the concepts introduced and used in this work.

Acknowledgement

I wish to thank Dr. Dorel Lucanu for carefully reading a preliminary version of this paper and correcting several errors.

Special thanks go to Professor Goguen who encouraged this work, and to Professor Futatsugi and the "CafeOBJ club" at JAIST who provided an excelent environment and useful feedback for this work.

2 Preliminaries

This work assumes certain familiarity from the reader's side with the basics of category theory, also some knowledge and experience with HSA and RWL might prove very useful.

Categorical notations and terminology generally follows [16], with the notably difference that we write composition diagramatically. Also, we may often use the subscript notation rather than the usual bracket notation when evaluating a function at an argument. When using 2-categories we stick with the standard terminology of 0-cells standing for (primitive) objects, 1-cells for arrows between objects, and 2-cells for arrows between arrows. We denote 1-cells by $\rightarrow$ and 2-cells by $\Rightarrow$; this also applies to the standard example of $\mathbb{C}at$, where 0-cells are categories, 1-cells are functors and 2-cells are natural transformations. The class of objects of a category $\mathbb{C}$ is, as usually, denoted as $|\mathbb{C}|$.

Since this work has its origins within the tradition of algebraic specification, we inevitably use a lot of algebraic specification notations and concepts, such as (many sorted) signature, algebras, homorphisms (our terminology and notations being consistent to [12]), but also the poweful concept of institution [11] which constitute the modern level of algebraic specification. We denote an institution $\mathfrak{I}$ by $(\mathbb{S}ign, \text{MOD}, Sen, \models)$, where $\mathbb{S}ign$ is the category of signatures, $\text{MOD}\colon \mathbb{S}ign \rightarrow \mathbb{C}at^{op}$ the model functor, $Sen\colon \mathbb{S}ign \rightarrow \mathbb{C}at$ the sentence functor, and $\models$ is the satisfaction relation. Given a signature morpshim ϕ the reduct functor $\text{MOD}(\phi)$ is often denoted as $_\restriction_\phi$ and we often overload ϕ as a short hand notation for $Sen(\phi)$.

The rest of the preliminary section is devoted to the brief presentation of HSA and RWL.

2.1 Behavioural Specification

Hidden sorted algebra was introduced in [12] as an algebraic semantics for the object paradigm capturing its main features such as local states, attributes, methods, classes, inheritance, concurrent distributed operation, etc. The central notion of the HSA algebra approach is that of *behavioural satisfaction*, that is we are interested in the behaviour of the objects rather than in the details of how they are implemented. HSA distinguishes between data, modelled with "visible" sorts, and states, modelled with "hidden" sorts. The recent paper [14] outlines a programme of research on HSA, also giving an overview of some results in this area. By **behavioural specification** we mean specification using HSA.

2.1.1 Hidden sorted algebra

In this section we provide the basic definition of HSA. For reasons of simplicity of presentation HSA assumes a fixed collection of data types given by a many sorted signature (V, Ψ) and a fixed (V, Ψ)-algebra. (V, Ψ, D) is called the **universe of data**. A **hidden sorted signature (over** (V, Ψ, D)**)** is a pair (H, Σ), where H is a set of **hidden** sorts, disjoint from V, and Σ is a $(H \cup V)$ signature, such that

(S1) if $\sigma \in \Sigma_{w,v}$ with $w \in V^*$ and $v \in V$, then $\sigma \in \Psi_{w,v}$; and

(S2) if $\sigma \in \Sigma_{w,v}$ then at most one element of w lies in H.

A **hidden sorted signature morphism** $\phi\colon (H, \Sigma) \to (H', \Sigma')$, is an ordinary signature morphism $\phi\colon \Sigma \to \Sigma'$ such that:

(M1) $\phi(v) = v$ for all $v \in V$ and $\phi(\sigma) = \sigma$ for all $\sigma \in \Psi$,

(M2) $\phi(H) \subseteq H'$, and

(M3) if $\sigma' \in \Sigma'_{w',s'}$ and some sort in w' lies in $\phi(H)$, then $\sigma' = \phi(\sigma)$ for some $\sigma \in \Sigma$.

A **hidden sorted model** over (H, Σ) is a Σ-algebra M such that $M{\restriction_\Psi} = D$ and a **homomorphism of hidden sorted models** $h\colon M \to M'$ is an ordinary homomorphism algebras such that $h{\restriction_\Psi} = 1_D$.

The reader might easily notice that this setup for HSA strongly resembles the setup for *constraint logic* of [6] in that (V, Ψ) can be regarded as a signature of "built-ins", D as a model of "built-ins", and Σ as an extension of (V, Ψ) with "logical" symbols. In this way (V, Ψ, D, Σ) appears as a constraint logic signature, thus enabling the direct use of the model and sentence functors of the constraint logic institution (see [6]) for dealing with variable rather than fixed data types.

2.1.2 Hiding and behaviour in institutions

Although behavioural specification is traditionally developed and studied within the context of equational logic (i.e., algebra), the essential core of the behavioural specification paradigm can be developed in a modern generic style [3], often referred as "institution-independent". In [3] we develop a generic

method for defining a behavioural satisfaction relation on top of any satisfaction relation in an institution; in particular, in this paper, we apply this methodolgy to the (mathematically complex) case of RWL.

The main idea of this approach is to provide a *semantic* definition of behavioural satisfaction as opposed to the syntactic traditional one. In the particular case of HSA, the two definitions coincide, but the former one has the advantage that is conceptually more general since it uses an operation on models rather than on sentences. Models usually have simple definitions while sentences sometimes might have complex definitions, moreover many institutions share the same notion of model, but they differ in the sentence part.

In order to describe the notion of behaviourally indistinguishable concisely [3] introduces *behaviour algebras* which are a simple kind of hidden sorted algebras but have a concise formulation of "behaviour". This is somewhat reminiscent of the Lawvere notion of algebra for an algebraic theory [17] in the sense that it is just a functor interpreting a *behaviour signature*, which is a conversion of a hidden sorted signature to a special category.[2] Hidden sorted algebras convert to behaviour algebras and back again and that there are final behaviour algebras. The morphism to the final algebra yields the required behavioural quotient. In [3] we use this to develop a way of taking an institution equipped with some extra data and producing an object version of the institution (or a "hidden" version). The extra data shows how the models of the institution can be viewed as representing behaviour models. Here is the idea in outline. We take an institution $\mathfrak{I}$ plus the extra data and produce an institution $\mathcal{H}(\mathfrak{I})$, thus

- We have a notion of $\mathcal{H}$ signature, and each $\mathcal{H}$ signature should have a corresponding $\mathfrak{I}$ signature;
- The models of the $\mathcal{H}$ signature are a subcategory of the models of the $\mathfrak{I}$ signature;
- For each model M of an $\mathcal{H}$ signature Σ we can derive a behaviour model;
- We compute an image of this behaviour model by taking the image factorisation of the unique morphism to the final behaviour model (in the particular case of HSA this corresponds to getting the quotient model by equating the elements which have the same observable behaviour);
- We convert this image model back into an $\mathcal{H}$ model $\overline{M}$.
- The satisfaction relation in $\mathcal{H}$ between the model M and any sentence e is defined as the satisfaction in $\mathfrak{I}$ between $\overline{M}$ and e.

All this should work smoothly when one changes signatures, so that we get an institution $\mathcal{H}$ equipped with an institution morphism to $\mathfrak{I}$. This is formalised by the main result of [3], and we use this methodology for defining the behavioural satisfaction in HSRWL.

[2] We will give extended and precise definitions of all these in Section 3.

2.2 *Rewriting Logic*

Rewriting logic was introduced by Meseguer [18] as a new unifying semantic theory for concurrency. In this paragraph we briefly review the basic concepts of RWL, but the reader is strongly advised to consult [18] for more details.

The syntax of rewriting logic is given by the **rewrite signatures**, which are algebraic theories (Σ, E), where Σ is an algebraic signature[3] and E is a collection of Σ-equations (called *structural equations*). The **sentences** of rewriting logic are conditional rewrite rules modulo E, i.e.,

$$(\forall X)\ [t] \to [t'] \textbf{ if } [u_1] \to [v_1] \ldots [u_k] \to [v_k]$$

where t, t', u_i, v_i are Σ-terms with variables X and modulo the equations in E. The left-hand side of **if** is the head of the rule and the right-hand side is the condition of the rule.

The **proof theory** of rewriting logic consists of 4 deduction rules strongly resembling the equational logic deduction rules, the crucial difference being the absence of the *symmetry* rule. This goes to the heart of the philosophy of RWL which is a logic about *changes* rather than equality.

Semantics of RWL is given by its models, called **(rewrite) systems** which provide with an elegant categorical formulation for the concept of concurrent distributed system. In [18] Meseguer presents an impressive list of examples from various areas of concurrency research. A model for the rewrite signature (Σ, E) is given by the interpretation of the corresponding algebraic theory into $\mathbb{C}at$. More concretely, a model A interprets each sort s as a category A_s, and each operation $\sigma \in \Sigma_{w,s}$ as a functor $\sigma_A \colon A_w \to A_s$, where A_w stands for $A_{s_1} \times \ldots \times A_{s_n}$ for $w = s_1 \ldots s_n$. Each Σ-term $t \colon w \to s$ gets a functor $t_A \colon A_w \to A_s$ by evaluating it for each assignment of the variables occuring in t with arrows from the corresponding carriers of A. The satisfaction of an equation $t = t'$ by A is given by $t_A = t'_A$;[4] in particular all structural equations should be satsified by A. A model morphism is a family of functors indexed by the sorts commuting the interpretations of the operations in Σ.

This algebra "enriched" over $\mathbb{C}at$ is a special case of *category-based equational logic* (see [4,5,13]) when letting the category $\mathbb{A}$ of models to be the interpretations of Σ into $\mathbb{C}at$ as abovely described, the category $\mathbb{X}$ of domains to be the category of many sorted sets, and the forgetful functor $\mathcal{U} \colon \mathbb{A} \to \mathbb{X}$ forgetting the interpretations of the operations and the composition between the arrows, i.e., mapping each category to its set of arrows. This enables the use of the machinery of category-based equational logic as a technical aide to the model theory of RWL.

The satisfaction of a rewrite rule $(\forall X)\ [t] \to [t']$ **if** $[u_1] \to [v_1] \ldots [u_k] \to [v_k]$ by a system A has a rather sophisticated definition using the concept of *subequaliser*. Let w be the string of sorts associated to the collection of variables X. Then

[3] Order sorted signature in the full generality, but for presentation simplicity we restrict ourselves to the many-sorted case.

[4] This definition extends without difficulty to conditional equations.

$$A \models (\forall X)\ [t] \to [t'] \textbf{ if } [u_1] \to [v_1] \ldots [u_k] \to [v_k]$$

iff there exists a natural transformation $J_A; t_A \Rightarrow J_A; t'_A$ where $J_A\colon Subeq((u_{iA}, v_{iA})_{i\in\overline{1,k}}) \to A_w$ is the subequaliser functor, i.e., the functor component of the final object in the category having pairs $(Dom(S) \xrightarrow{S} A_w, (S; u_{iA} \overset{\alpha_i}{\Rightarrow} S; v_{iA})_{i\in\overline{1,k}})$ as objects and functors H such that $H; S' = S$ and $H\alpha' = \alpha$ as arrows.

The satisfaction relation between the sentences and the syntax of rewriting logic gives an institution in which the signature morphisms are morphisms of algebraic theories and the translation of the quantifiers and terms (modulo structural axioms) along the signature morphisms is the same as in the institution of equational logic modulo axioms (denoted *ELM* in [7]).

3 (2-)Behaviour Signatures and Systems

Behaviour systems provide a concise formulation of "behaviour" for concurrent systems by converting a hidden sorted rewrite model (system) into another kind of system which is mathematically simpler. Behaviour systems generalise behaviour algebras of [3] in the same way the models of rewriting logic generalise the models of equational logic, i.e., general algebra.

Definition 3.1 *A* **(2-)behaviour signature** *is a (2-)category with distinguished 0-cell d such that there are no 1-cells whose source is d except the identity; we let a* **morphism of (2-)behaviour signatures** *be a (2-)functor preserving the distinguished 0-cell and such that distinguished 0-cells are the only 0-cells mapped to distinguished 0-cells.*

We write a (2-)behaviour signature as a pair (C, d) where C is a (2-)category and d is its distinguished 0-cell. We use h and h' for 0-cells different from d, and we use e for 1-cells.

Notice that 2-behaviour signatures are simple behaviour signatures by ignoring the 2-cells. Similarly for signature morphisms.

Fact 3.2 *Each hidden sorted signature (H, Σ) over (V, Ψ, D) gives rise to a 2-behaviour signature (C, d) where*

- $|C| = H \cup \{d\}$; *and*
- *C is freely generated by the family of sets of 2-cells $C[h, n] = \{\langle \sigma, a\rangle \overset{r}{\Rightarrow} \langle \sigma, a'\rangle \mid \sigma \in \Sigma_{vhv',n}, v, v' \in V^*, r \in D_{vv'}(a, a')\}$ where $h \in H$ and $n \in H \cup \{d\}$.*

Each hidden sorted signature morphism $\phi\colon (H, \Sigma) \to (H', \Sigma')$ gives rise to a 2-behaviour signature morphism ϕ by letting $\phi(\langle \sigma, a\rangle \overset{r}{\Rightarrow} \langle \sigma, a'\rangle) = \langle \phi(\sigma), a\rangle \overset{r}{\Rightarrow} \langle \phi(\sigma), a'\rangle$.

In the following we provide (2)-functorial semantics for behaviour systems in the style of Lawvere functorial semantics for algebraic theories [17].

Definition 3.3 *Let (C, d) be a (2-)behaviour signature and D be a category. A* **(2-)behaviour system** *over (C, d, D) is a (2-)functor $A\colon C \to \mathbb{C}at$ such that $A_d = D$. A morphism of (2-)behaviour systems is a (2-)natural transformation*

such that its component at d is the identity on D. By $\mathbb{B}Sys(C,d,D)$ we denote the category of behaviour systems and their morphisms and by $2\mathbb{B}Sys(C,d,D)$ the category of 2-behaviour systems and 2-behaviour morphisms.

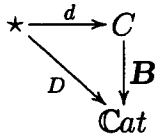
3.1 The Final Behaviour System

As emphasised in [3,12], final models play a crucial role in the semantics of behavioural specification. Following [3], in Section 4.2 we use final models for providing a semantic definition for behavioural satisfaction in RWL. We concentrate here on the final behaviour system (in both the simple and the more complex version) which we obtain as a Kan-extension.

Theorem 3.4 *For any (2-)behaviour signature (C,d,D) there exists the final (terminal) (2-)behaviour system $\boldsymbol{B}$ which can be obtained as the right (2-)Kan extension of d along D.*

Proof. If d and D are regarded as (2-)functors from the final category $\star$ consisting of one identity cell, then the final (2-)behaviour system over (C,d,D) is just the right (2-)Kan extension of d along D.

Let's first discuss the simpler case of ordinary behaviour systems. The right Kan-extension exists since $\mathbb{C}at$ is small complete, i.e., it has all small limits (Corrolary 2 pg. 235 of [16]).



A slightly subtle point concerns the equality $d;\boldsymbol{B} = D$ for $\boldsymbol{B}$ the right-Kan extension of d along D. This is equivalent with the condition that $\boldsymbol{B}$ is really a behaviour system, i.e., $\boldsymbol{B}(d) = D$. This follows directly by Corollary 3 pg. 235 of [16] because d is full (since there are no arrows in C out of d except the identity) and faithful as a functor.

The 2-case can be treated similarly by using (the dual of) Theorem 6.7.7 from [2] [5] applied to the $\mathbb{C}at$-enriched case, for example.

For understanding the "behavioural" structure of $\boldsymbol{B}$, we need to explicitate its construction (see Corollary 2 pg. 235 of [16] for the case of ordinary categories):

Corollary 3.5 *The structure of the final 2-behaviour system $\boldsymbol{B}$ is given by:*

(i) *for each 0-cell $n \in |C|$, $\boldsymbol{B}_n$ is the functor category $D^{C(n,d)}$,*

(ii) *for each 1-cell $e \in |C(n,n')|$, $\boldsymbol{B}_e$ is $D^{C(e,d)}$, i.e., the functor $D^{C(n,d)} \to D^{C(n',d)}$ given by $(\boldsymbol{B}_e)(f)_c = f_{e;c}$ where $f \in D^{C(n,d)}$ and $c \in C(n',d)$, and*

(iii) *(**for the 2-case only**) for each 2-cell $e \stackrel{r}{\Rightarrow} e'$, $\boldsymbol{B}_r$ is $D^{C(r,d)}$, i.e., the natural transformation $D^{C(e,d)} \Rightarrow D^{C(e',d)}$ given by $((\boldsymbol{B}_r)_f)_c = f_{rc}$ for each $f \in D^{C(n,d)}$ and each $c \in C(n',d)$.*

[5] Generalising the standard Kan-extension existence result to enriched category theory.

Notice that the behaviour system underlying the final 2-behaviour system is not final! One can easily notice this when comparing the definition of $\boldsymbol{B}_n$ (for n arbitrary 0-cell) in each case; in the 2-case this is a proper functor category while in the simple case it is just a power category (having of course "more" objects and arrows). This difference finally amounts in principle to different concepts of behavioural satisfaction for RWL.

3.2 The Image Behaviour System

We now turn to the definition of image behaviour which plays the most important rôle for the definition of behavioural satisfaction. In the case of RWL this provides a non-trivial generalisation of HSA behavioural quotients because in RWL the behaviour image system apart of equating elements and transitions under the behavioural equivalence relation, also adds new transitions to the original model. This is formally achieved by using a rather abstract formulation, the notion of *image factorisation system* as defined in [1]. In fact our factorisation system for behaviour systems is inherited from one of the factorisation systems in $\mathbb{C}at$ as follows:

Fact 3.6 *The category of systems over (C, d, D) has a canonical image factorisation system $(\mathcal{E}_C, \mathcal{M}_C)$ with $\mathcal{E}_C = \{e$ morphism of (2-)behaviour systems $\mid e_h$ surjective on objects for any $h\}$ and $\mathcal{M}_C = \{m$ morphism of (2-)behaviour systems $\mid m_h$ full, faithful and injective on objects for any $h\}$.*

Proof. Each component of a morphism of (2-)behaviour systems factors in $\mathbb{C}at$ through the image factorisation system $(\mathcal{E}, \mathcal{M})$ of $\mathbb{C}at$ where $\mathcal{E} = \{e \mid e$ surjective on objects$\}$ and $\mathcal{M} = \{m \mid m$ full, faithful and injective on objects$\}$, then we use the Diagonal Fill-in Property for image factorisation systems to define the image behaviour system on arrows.

More specifically, suppose $m\colon A \to \boldsymbol{B}$ is a morphism of systems and $e\colon n \to n'$ a 1-cell in C, so that $A_e\colon A_n \to A_{n'}$ and $\boldsymbol{B}_e\colon \boldsymbol{B}_n \to \boldsymbol{B}_{n'}$. Then we factorise $m_n\colon A_n \to \boldsymbol{B}_n$ to get an intermediate $\overline{A}_n$, similarly for n'. We then use the fill in property to get $\overline{A}_e\colon \overline{A}_n \to \overline{A}_{n'}$. This gives the factorisation of m with image system $\overline{A}$.

Only for the case of of 2-behaviour systems, we also need to define $\overline{A}_r$ for any 2-cell $e \stackrel{r}{\Rightarrow} e'$. This is given by simply restricting $\boldsymbol{B}_r$.

Definition 3.7 *Given a behaviour system A, its* **image (2-)behaviour system** *is $\overline{A}$, where*

$$A \longrightarrow \overline{A} \longrightarrow \boldsymbol{B}$$

is the factorisation of the unique behaviour (2-)system morphism $A \to \boldsymbol{B}$, where $\boldsymbol{B}$ is the final (2-)behaviour system.

The final semantics concept of image behaviour system (or model) is dual to the concept of *reachable submodel* from the initial semantics. Reachable submodels can be obtained in the same way by factoring the unique morphism from the initial model to the corresponding model.

Any morphism $\phi\colon (C, d) \to (C', d)$ of behaviour signatures determines a functor

$-\restriction_\phi : \mathbb{B}Sys(C', d, D) \to \mathbb{B}Sys(C, d, D)$ mapping a behaviour system A' to $\phi; A'$ and any morphism f' of behaviour systems to $\phi f'$ (i.e., the vertical composition between ϕ as a functor and f' as a natural transformation).

Corollary 3.8 *For any morphism $\phi : (C, d) \to (C', d)$ of (2-)behaviour signatures, $\restriction_\phi$ is a morphism of factorisation systems $(\mathcal{E}_{C'}, \mathcal{M}_{C'}) \to (\mathcal{E}_C, \mathcal{M}_C)$, i.e., $\mathcal{E}_{C'}\restriction_\phi \subseteq \mathcal{E}_C$ and $\mathcal{M}_{C'}\restriction_\phi \subseteq \mathcal{M}_C$.*

Lemma 3.9 *Let $\Phi : I \to J$ be a functor surjective on objects and D be any category. Then the functor $D^\Phi : D^J \to D^I$ is faithful and injective on objects.*

Corollary 3.10 *Let $\phi : (C, d) \to (C', d)$ be a morphism of (2-)behaviour signatures with $\phi(h, d) : C(h, d) \to C'(\phi(h), d)$ surjective for any h. Let $\mathbf{B}$ be the final (2-)behaviour system over (C, d, D) and $\mathbf{B}'$ be the final system over (C', d, D). Then the unique morphism of (2-)behaviour systems $m : \mathbf{B}'\restriction_\phi \to \mathbf{B}$ is injective in all components, i.e., it belongs to $\mathcal{M}_C$.*

Proof. Fix $h \in |C| - \{d\}$. $(\mathbf{B}'\restriction_\phi)_h = \mathbf{B}'_{\phi(h)} = D^{C'(\phi(h),d)}$. Then by applying the previous lemma for $\phi(h, d)$ in the rôle of Φ, we get that m_h is faithful and injective on objects, therefore $m \in \mathcal{M}_C$.

Fact 3.11 *Any morphism of (2-)behaviour signatures generated by a hidden sorted signature morphism has the surjectivity property of the previous Corollary.*

Proof. By using the condition (M3) of the definition of morphisms of hidden sorted signatures.

4 Hidden Sorted Rewriting Logic

In this section we use the technique previously developed for defining the basic ingredients of HSRWL, such as signatures, models, and satisfaction of sentences by models. We finally show that HSRWL is an institution.

4.1 Signatures and Models

Definition 4.1 *A* **hidden sorted rewrite signature** *is given by (H, Σ, E), where (H, Σ) is a hidden sorted signature over (V, Ψ, D) with the difference that D is a rewrite model for (V, Ψ) rather than an algebra, and E is a collection of Σ-equations. A* **morphism** *$\phi : (H, \Sigma, E) \to (H', \Sigma', E')$ of hidden sorted rewrite signatures is a morphism of hidden sorted signatures $(H, \Sigma) \to (H', \Sigma')$ such that the following is satisfied:*

(M4) $\phi(E) \models_{\Sigma'} E'$

A **hidden sorted rewrite model** *M for a hidden sorted rewrite signature (H, Σ, E) over (V, Ψ, D) is just a (Σ, E)-rewrite model such that $M\restriction_\Psi = D$.*

Condition (M4) corresponds to the "encapsulation of structural axioms" and together with (M3) play the crucial rôle in obtaining the Satisfaction Condition for HSRWL.

Fact 4.2 *Given a hidden sorted rewrite model M for a signature (H, Σ, E), one can canonically extract a 2-behaviour system $[M]$ for the 2-behaviour signature (C, d) generated by (H, Σ), by letting*

- $[M]_h = M_h$ *for each* $h \in H$,
- $[M]_{\langle \sigma, a \rangle} = \sigma_M(_, a) : M_h \to M_{h'}$ *for each* $\sigma \in \Sigma_{vhv',h'}$ *and* $a \in |D_{vv'}|$, *and*
- $([M]_r(b) = \sigma_M(r, 1_b)$ *for each* $r \in D_{vv'}(a, a')$ *and* $b \in |M_h|$.

This can be easily extended to a forgetful functor $[_]$ from the category of hidden sorted rewrite models to the category of 2-behaviour systems.

Lemma 4.3 *Let A be a (H, Σ)-rewrite model and $m : [A] \to B$ be a morphism of behaviour systems. There is an unique morphism of (H, Σ)-rewrite models $m^\sharp : A \to B^\sharp$ such that $[m^\sharp] = m$.*

Proof. Because $[m^\sharp]$ should be m, for any $h \in H$ we take $B^\sharp_h$ to be B_h and $m^\sharp_h$ to be m_h. If $\sigma \in \Sigma_{vhv',h'}$, then, for all $a \in |D_v|$, $b \in B_h$, $\sigma_{B^\sharp}(b, a)$ is $B_{\langle \sigma, a \rangle}(b)$. If $\sigma \in \Sigma_{v,h}$, $v \in V^*$, $h \in H$, then, for all $a \in D_v$, $\sigma_{B^\sharp}(a)$ is defined as $m_h(\sigma_A(a))$. These define a hidden sorted rewrite model $B^\sharp$ and a morphism $m^\sharp : A \to B^\sharp$. Notice that $m^\sharp$ is indeed a morphism of hidden sorted models because of the naturality of m and of the definition of the interpretations of the operations $\sigma \in \Sigma_{v,h}$, $v \in V^*$, $h \in H$, in $B^\sharp$.

4.2 Behavioural Satisfaction for Rewriting Logic

In this section we define the behavioural satisfaction relation for HSRWL by using the behaviour image system rather than the 2-behaviour system. 2-behavioural satisfaction is discussed in a section below.

Definition 4.4 *Let (H, Σ, E) be a hidden sorted signature. Then for each rewrite model M, its* **behaviour image** *$\overline{M}$ is defined as $\overline{[M]}^\sharp$, where*

- *$\overline{[M]}$ is the image of the unique morphism of behaviour systems $[M] \to \boldsymbol{B}$, and*
- *$\overline{[M]}^\sharp$ is obtained by lifting the canonical map $[M] \to \overline{[M]}$ back to (H, Σ)-rewrite models.*

A (H, Σ, E)-rewrite model M **behaviourally satisfies** *a sentence $[\rho]$ iff $\overline{M} \models \rho$. We write this as $M \models_{(H,\Sigma,E)} [\rho]$ (or just $M \models [\rho]$ for short).*

Sentences $[\rho]$ in this case can be either (possibly conditional) rules or equations modulo the structural equations E. The notation $[\rho]$ extends the usual notation for equivalence classes of terms modulo E to rules and equations in the obvious way.

Theorem 4.5 [Satisfaction Condition for HSRWL] *Let $\phi : (H, \Sigma, E) \to (H', \Sigma', E')$ be a morphism of hidden sorted rewrite signatures, M' be a (H', Σ', E')-rewrite model, and ρ be a Σ-rule or a Σ-equation. Then*

$$M' \models_{(H',\Sigma',E')} \phi([\rho]) \quad \textit{iff} \quad M'\restriction_\phi \models_{(H,\Sigma,E)} [\rho]$$

Proof. This proof follows the idea of the proof of Theorem 15 of [3]. Since there is no danger of confusion by ϕ we will denote both the hidden sorted signature morphism $(H, \Sigma) \to (H', \Sigma')$ and the corresponding behaviour signature morphism.

We first show the naturality of the behaviour image wrt the signature morphisms, i.e., that

$$\overline{M'}\restriction_\phi = \overline{M'\restriction_\phi}$$

Consider the following diagram of behaviour systems:

$$\begin{array}{ccccc} \phi;[M'] & \xrightarrow{e^1} & \phi;\overline{[M']} & \xrightarrow{m^1} & \phi;\boldsymbol{B}_{\Sigma'} \\ \| & & & & \downarrow m \\ [M'\restriction_\phi] & \xrightarrow[e^2]{} & \overline{[M'\restriction_\phi]} & \xrightarrow[m^2]{} & \boldsymbol{B}_\Sigma \end{array}$$

We have that $e^1 \in \mathcal{E}_C$ and $m^1 \in \mathcal{M}_C$ by Corollary 3.8, $e^2 \in \mathcal{E}_C$ and $m^2 \in \mathcal{M}_C$ by definition, and $m \in \mathcal{M}_C$ by Corollary 3.10. The uniqueness of the factorisation gives us $\phi;\overline{[M']} = \overline{[M'\restriction_\phi]}$ (modulo a canonical isomorphism). By applying Lemma 4.3 for $e^1 = e^2$, we get that $\overline{M'}\restriction_\phi = \overline{M'\restriction_\phi}$.

Now we can proceed with the proof of the Satisfaction Condition. We may assume that ρ is a rule, the case when ρ is an equation can be treated similarly and it might be more familiar from the HSA.

$$\begin{array}{lll} M' \models_{(H',\Sigma',E')} \phi([\rho]) & \text{iff} & M' \models_{\Sigma'} E' \text{ and } \overline{M'} \models_{\Sigma'} \phi(\rho) \\ & \text{iff} & M'\restriction_\phi \models_\Sigma E \text{ and } \overline{M'} \models_\Sigma \phi(\rho) \\ & & \text{((M4) and the CBEL Satisfaction Condition)} \\ & \text{iff} & M'\restriction_\phi \models_\Sigma E \text{ and } \overline{M'}\restriction_\phi \models_\Sigma \rho \\ & & \text{(RWL Satisfaction Condition)} \\ & \text{iff} & M'\restriction_\phi \models_\Sigma E \text{ and } \overline{M'\restriction_\phi} \models_\Sigma \rho \\ & & \text{(by the previous argument)} \\ & \text{iff} & M'\restriction_\phi \models_{(H,\Sigma,E)} [\rho] \\ & & \text{(by definition).} \end{array}$$

4.3 Towards a Proof Theory for RWL Behavioural Satisfaction

In this section we develop some syntactic characterisations of the behavioural satisfaction relation defined in the previous section. This opens the door for using recent advanced techniques developed for HSA (such as the "coinduction" of [14]) for supporting proofs of properties of behavioural specification in RWL.

Proposition 4.6 *Let (H, Σ, E) be a hidden sorted rewrite signature. Then for each (H, Σ, E)-model M and for any rule $[t] \to [t']$,*

$M \models\!\models [t] \to [t']$ *implies* $M \models c(t) \to c(t')$ *for all contexts* c *of visible sort*

Proof. Given a hidden sort $h \in H$, there is a canonical one-one correspondence

$$\{c(z) \mid c \text{ context of visible sort with the variable } z \text{ of sort } h\} = C(h,d)$$

where (C,d) is the behaviour signature determined by (H,Σ,E).

$M \models\!\models [t] \to [t']$ means that $\overline{M} \models t \to t'$, which means there exists a natural transformation $\alpha\colon t_{\overline{M}} \Rightarrow t'_{\overline{M}}$. We need to find a natural transformation $\alpha^c\colon c(t)_M \Rightarrow c(t)_M$ for each visibile context $c \in C(h,d)$. Assume $t,t'\colon w \to h$. Then for each $p \in |M_w|$ we define

$$\alpha^c_p = c_{\overline{M}}(\alpha_{\overline{p}}),$$

where $\overline{p}$ is the image of p in $\overline{M}_w$ via the canonical map $M \to \overline{M}$. We have to show the naturality of α^c_p, i.e., the commutativity of the following diagram:

$$\begin{array}{ccc} p & c_M(t_M(p)) \xrightarrow{\alpha^c_p} & c_M(t'_M(p)) \\ {\scriptstyle\varphi}\downarrow & {\scriptstyle c_M(t_M(\varphi))}\downarrow & \downarrow{\scriptstyle c_M(t'_M(\varphi))} \\ p' & c_M(t_M(p')) \xrightarrow[\alpha^c_{p'}]{} & c_M(t'_M(p')) \end{array}$$

for all $\varphi \in M_w$. But this follows because c has visible sort, which means that $c_M(t_M(\varphi)) = c_{\overline{M}}(t_{\overline{M}}(\overline{\varphi})) \in D$, and by the naturality of α.

Theorem 4.7 *Let (H,Σ,E) be a hidden sorted rewrite signature over (V,Ψ,D) with D partial order. Then for each (H,Σ,E)-model M and for any rule $[t] \to [t']$,*

$M \models\!\models [t] \to [t']$ *iff* $M \models c(t) \to c(t')$ *for all contexts* c *of visible sort*

Proof. Assume $M \models c(t) \to c(t')$. By definition $M \models\!\models [t] \to [t']$ is equivalent to $\overline{M} \models t \to t'$. So we have to find a natural transformation $\overline{\alpha}\colon t_{\overline{M}} \Rightarrow t'_{\overline{M}}$.

Assume that $t,t'\colon w \to h$, and as in Proposition 4.6 notice the one-one correspondence between the contexts of argument h and $C(h,d)$. So, by hypothesis we know that for each $c \in C(h,d)$, there exists $\alpha^c\colon c(t)_M \to c(t')_M$.

In the virtue of the definition of $\overline{M}$ we may consider $\overline{M}_h$ as a full subcategory of $\boldsymbol{B}_h$. The for each $p \in |M_h|$ we define

$$\overline{\alpha}_{\overline{p}} = \{\alpha^c_p\}_{c \in C(h,d)}$$

The correctenss of this definition for $\overline{\alpha}\colon t_{\overline{M}} \Rightarrow t'_{\overline{M}}$ is assured by the fact that because $\overline{M}_h \subseteq \boldsymbol{B}_h$ is full, the canonical map $M \to \overline{M}$ is surjective on the elements (i.e., objects) of the carriers and it is a congruence, therefore for each $c \in C(h,d)$, if $\overline{p} = \overline{p_1}$ then α^c_p and $\alpha^c_{p_1}$ have the same source and target in D, so they are equal.

We still have to prove the naturality of $\overline{\alpha}$, i.e., the commutativity of

$$\begin{array}{ccc} \overline{p} & t_{\overline{M}}(\overline{p}) & \xrightarrow{\overline{\alpha}_{\overline{p}}} t'_{\overline{M}}(\overline{p}) \\ \overline{\varphi}\downarrow & {\scriptstyle t_{\overline{M}}(\overline{\varphi})}\downarrow & \downarrow{\scriptstyle t'_{\overline{M}}(\overline{\varphi})} \\ \overline{p'} & t_{\overline{M}}(\overline{p'}) & \xrightarrow[\overline{\alpha}_{\overline{p'}}]{} t'_{\overline{M}}(\overline{p'}) \end{array}$$

in $\overline{M}_h$ for all $\overline{\varphi} \in \overline{M}_w(\overline{p}, \overline{p'})$. If we consider this diagram in $\boldsymbol{B}_h$, then it commutes componentwise, i.e.,

$$\begin{array}{ccc} \overline{p} & c_M(t_M(p)) & \xrightarrow{\alpha^c_p} c_M(t'_M(p)) \\ \overline{\varphi}\downarrow & {\scriptstyle c_{\overline{M}}(t_{\overline{M}}(\overline{\varphi}))}\downarrow & \downarrow{\scriptstyle c_{\overline{M}}(t'_{\overline{M}}(\overline{\varphi}))} \\ \overline{p'} & c_M(t_M(p')) & \xrightarrow[\alpha^c_{p'}]{} c_M(t'_M(p')) \end{array}$$

commutes for each $c \in C(h,d)$ because in D there is at most one arrow $c_M(t_M(p)) \to c_M(t'_M(p'))$.

Corollary 4.8 *For any hidden sorted rewrite signature (H,Σ,E) over (V,Ψ,D) with D partial order, any model M, and any rule $[t] \to [t']$, we have that*

$$M \models\!\!\!\models [t] \to [t'] \;\; \textit{if} \;\; M \models t \to t'$$

Proof. By the soundness of RWL $M \models [t] \to [t']$ implies that $M \models c(t) \to c(t')$ for all contexts c of visible sort, which further implies $M \models\!\!\!\models [t] \to [t']$ by Theorem 4.7.

Corollary 4.9 *Let (H,Σ,E) be a hidden sorted rewrite signature over (V,Ψ,D) with D partial order and $[R]$ be a rewrite theory. Then*

$$[R] \models\!\!\!\models [t] \to [t'] \;\; \textit{iff} \;\; [R] \vdash [c(t)] \to [c(t')] \;\; \textit{for all contexts} \;\; c \;\; \textit{of visible sort}$$

Proof. By the completeness of RWL, $[R] \vdash [c(t)] \to [c(t')]$ is equivalent to $[R] \models [c(t)] \to [c(t')]$.

First assume $[R] \models\!\!\!\models [t] \to [t']$ and consider a (H,Σ,E)-model for R. By Corollary 4.8, $M \models\!\!\!\models [R]$, therefore $M \models\!\!\!\models [t] \to [t']$, which by Proposition 4.6 implies $M \models c(t) \to c(t')$.

For the converse assume $[R] \models [c(t)] \to [c(t')]$ and consider a model M such that $M \models\!\!\!\models [R]$. Then $\overline{M} \models R$, which implies $\overline{M} \models c(t) \to c(t')$, which means $M \models\!\!\!\models [c(t)] \to [c(t')]$. But since c is of visible sort, this is the same with $M \models c(t) \to c(t')$ which by Theorem 4.7 implies $M \models\!\!\!\models [t] \to [t']$.

5 2-Behavioural Satisfaction

By using 2-behaviour system instead of behaviour systems, we can re-use Definition 4.4 for defining **2-behavioural satisfaction**, which we denote by $\models\!\!\!\models^2$. Similarly to Theorem 4.5 the ordinary behavioural satisfaction case, we have a Satisfaction Condition for $\models\!\!\!\models^2$:

Theorem 5.1 *Let $\phi\colon (H,\Sigma,E) \to (H',\Sigma',E')$ be a morphism of hidden sorted rewrite signatures, M' be a (H',Σ',E')-rewrite model, and ρ be a Σ-rule or a Σ-equation. Then*

$$M' \models^2_{(H',\Sigma',E')} \phi([\rho]) \quad \textit{iff} \quad M'\restriction_\phi \models^2_{(H,\Sigma,E)} [\rho]$$

The rest of this section is devoted to the study of the relationship betwen behavioural satisfaction and the 2-behavioural satisfaction relation. Fix a signature (H,Σ,E) over (V,Ψ,D), and let $\boldsymbol{B}$ denote the final behaviour system, $\boldsymbol{B}^2$ the final 2-behaviour system.

For any model M, let $\overline{M}$ denote its image behaviour system and $\overline{\overline{M}}$ denote its image 2-behaviour system.

Definition 5.2 *Let $h \in H$. Then $a, a' \in M_h$ are* **behaviourally equivalent** *iff $\overline{a} = \overline{a'}$, i.e., they get identified by the canonical map $M \to \overline{M}$; we denote this by $a \sim a'$.*

Similarly, a and a' are **2-behaviourally equivalent** *iff $\overline{\overline{a}} = \overline{\overline{a'}}$; we denote this by $a \approx a'$.*

Fact 5.3 *Both $\sim$ and $\approx$ are Σ-congruences.*

Lemma 5.4 *Let $h \in H$ and let $a \xrightarrow{\varphi} b, a' \xrightarrow{\varphi} b' \in M_h$. Then,*

(i) *$a \sim a'$ iff $[M]_c(a) = [M]_c(a')$ for all $c \in |C(h,d)|$,*

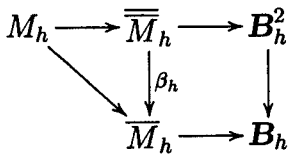
(ii) *$a \approx a'$ iff $a \sim a'$ and $([M]_r)_a = ([M]_r)_{a'}$ for all $c \xRightarrow{r} c' \in C(h,d)$,*

(iii) *$\varphi \sim \varphi'$ iff $[M]_c(\varphi) = [M]_c(\varphi')$ for all $c \in |C(h,d)|$, and*

(iv) *$\varphi \approx \varphi'$ iff $a \approx a', b \approx b'$, and $\varphi \sim \varphi'$.*

Proof. By explicitating the action of the unique (2-) behaviour system morphism from $[M]$ to $\boldsymbol{B}$ (or $\boldsymbol{B}^2$ in the 2-case).

Proposition 5.5 *There exists an unique morphism of hidden sorted rewrite models $\beta\colon \overline{\overline{M}} \to \overline{M}$ such that for each $h \in H$ the following diagram commutes in $\mathbb{C}at$:*



where $\boldsymbol{B}^2 \to \boldsymbol{B}$ is the unique behaviour system morphism from $\boldsymbol{B}^2$ (regarded as a behaviour system) to $\boldsymbol{B}$.

Proof. We define $\beta_h(\overline{\overline{a}}) = \beta_h(\overline{a})$ for each $a \in M_h$. This covers the definition of β_h on objects and arrows originating from M_h. Let now $\overline{\overline{a}} \xrightarrow{\varphi} \overline{\overline{a'}}$ such that there is no arrow $a \xrightarrow{\theta} a'$ in M_h with $\overline{\overline{\theta}} = \varphi$. then because $\overline{\overline{M}}_h \subseteq \boldsymbol{B}^2_h$ is full, $\varphi \in \boldsymbol{B}^2_h$. Let φ' be its image in $\boldsymbol{B}_h$ via the canonical map $\boldsymbol{B}^2_h \to \boldsymbol{B}_h$. Since $\overline{M}_h$ is full, $\varphi' \in \overline{M}_h$, therefore we may define $\beta_h(\varphi) = \varphi'$. Routine verifications will show that β_h is functor and that β is a model morphism.

Corollary 5.6 *Using the same notations as above,*

- *β is full (in all components) iff $\boldsymbol{B}^2_h \to \boldsymbol{B}_h$ is full for each $h \in H$, and*

- *β is isomorphism iff $\mathbf{B}^2_h \to \mathbf{B}_h$ is a full subcategory for each $h \in H$.*

Corollary 5.7 *If $\mathbf{B}^2_h \to \mathbf{B}_h$ is full for each $h \in H$, then for each hidden sorted rewrite model M and each unconditional equation or rewrite rule $[\rho]$,*

$$M \models [\rho] \;\; \textit{if} \;\; M \models^2 [\rho]$$

Proof. In the virtue of the definition of behavioural satisfaction (either in the simple case and in the 2-case), we have to prove that $\overline{M} \models \rho$ if $\overline{\overline{M}} \models \rho$.

If $[\rho]$ is an equation, then it is easy to check that by using the classical argument that quotients preserve the validity of unconditional equations. This is due to the fact that $\beta\colon\ \overline{\overline{M}} \to \overline{M}$ is a surjective on objects and full homomorphism.

If ρ is a rewrite rule $t \to t'$, assume $\overline{\overline{M}} \models \rho$. Then there exists $\overline{\overline{\alpha}}\colon\ t_{\overline{\overline{M}}} \Rightarrow t'_{\overline{\overline{M}}}$. We define $\overline{\alpha} = \beta(\overline{\overline{\alpha}})$ and because β is full and surjective on objects and $\overline{\overline{\alpha}}$ is natural, we can easily prove that $\overline{\alpha}$ is natural too.

Directly from the second part of Corollary 5.6 we deduce the following:

Corollary 5.8 *If $\mathbf{B}^2_h \to \mathbf{B}_h$ is full for each $h \in H$, then for each model M and each sentence $[\rho]$,*

$$M \models [\rho] \;\; \textit{iff} \;\; M \models^2 [\rho]$$

Corollary 5.9 *If the system of data D is a partial order, then for each model M and each sentence $[\rho]$*

$$M \models [\rho] \;\; \textit{iff} \;\; M \models^2 [\rho]$$

6 Conclusions and Future Research

We internalised behavioural satisfaction to RWL by using a semantic defintion of behavioural satisfaction between hidden sorted rewrite models on one side, and equations and rules on the other side. This gives an instituion for HSRWL which generalises both HSA and RWL, thus providing a unitary logic underlying the integration of the two specification paradigms.

This paper concentrates on the semantic and logical foundations of this new paradigm, but also makes some links to the proof theory. We feel that further work needs to be done in the following areas:

- exploration of advanced techniques for proving behavioural properties of concurrent distributed systems by shifting to the HSRWL the techniques already developed for HSA, such as the "coinduction" of [14],
- exploration of relevant examples in connection to the above,
- clarify the relationship between the two different treatments in HSRWL of the object paradigm inherited from HSA and RWL, and
- further study of the relationship between behavioural satisfaction and 2-behavioural satisfaction.

We also plan to use HSRWL as a framework for defining the formal semantics of the CafeOBJ system [9] which actaully implements both specification

paradigms. In fact CafeOBJ was the first driving force behind this research.

A An example

We devote this appendix for gradually illustrating the concepts introduced in this paper by using the (simple) example of a behavioural specification in HSRWL of a non-deterministic counter (the code is written in CafeOBJ [9]).

The data
As data we consider the natural numbers with non-deterministic choice as specified by the following RWL module importing a library of natural numbers.

```
module NAT? {
  extending (NAT)
  op _?_ : Nat Nat -> Nat
  vars M N : Nat
  rule M ? N => M .
  rule M ? N => N .
}
```

This defines a "visible" signature (V, Ψ). For the model D of data we consider $|D_{\mathtt{Nat}}| = \mathcal{P}_f\omega$, where $\mathcal{P}_f\omega$ is the set of all finite sets of natural numbers. $D_{\mathtt{Nat}}(S, S') = \emptyset$ if $S' \not\subseteq S$ and $D_{\mathtt{Nat}}(S, S') = \{S \to S'\}$ if $S' \subseteq S$. We define the interpretation of the "choice" operator ? as $S?S' = S \cup S'$ and the interpretations of the usual operations on the natural numbers in a straighforward manner, for example

$$S + S' = \{x + x' \mid x \in S, x' \in S'\} \text{ for all } S, S' \in \mathcal{P}_f\omega$$

Notice that the interpretations of the operations are monotonic wrt inclusion between sets of natural numbers, so they are functors.

The counter: signature and models
Now we can use the data module NAT? for specifying the counter object.

```
module COUNTER {
  protecting (NAT?)
  [ Counter ]                          -- class (hidden sort)
  op add : Nat Counter -> Counter      -- method
  op read : Counter -> Nat             -- attribute
  var N : Nat
  var C : Counter
  eq read(add(N,C)) = N + read(C) .  -- structural equation
}
```

This defines a hidden sorted rewrite signature (H, Σ, E) over (V, Ψ, D) where H consists of only one hidden sort (Counter) and Σ contains one method (add) and one attribute (read), and E contains one structural equation.

We consider two models for the module COUNTER. The first one, denoted A, is a "history" model that interprets $A_{\mathtt{Counter}}$ as the *set* $(\mathcal{P}_f\omega)^*$ of lists of finite

sets of natural numbers. In other words in this model the states of the counter are implemented as the lists $[S_1 \dots S_k]$ with $S_i \subseteq \omega$ finite for $i \in \overline{1,k}$. This is a static implementation in the sense that there are no transitions between the states of the counter. $\mathtt{add}_A$ just adds new sets to the history list, and $\mathtt{read}_A$ evaluates the state by $\mathtt{read}_A([S_1 \dots S_k]) = S_1 + \dots + S_k$.

The second model, denoted $\overline{A}$ (which we will later see that is in fact the bahaviour image of the model A), implements $\overline{A}_{\mathtt{Counter}}$ as $D_{\mathtt{Nat}}$, $\mathtt{add}_{\overline{A}}(S,C) = S + C$, and $\mathtt{read}_{\overline{A}}(C) = C$. Notice that both models satisfy the structural equation.

The counter: behaviour signature and systems; final behaviour system

The hidden sorted signature (H, Σ) determines a behaviour signature C where $|C| = \{\mathtt{Counter}, d\}$. Because we have only one method and one (unparameterised by data) attribute, C is the category freely generated by the set of arrows $C[\mathtt{Counter}, \mathtt{Counter}] = \mathcal{P}_f\omega$ and $C[\mathtt{Counter}, d]$ having only one element. This means that both $C(\mathtt{Counter}, \mathtt{Counter})$ and $C(\mathtt{Counter}, d)$ consists of lists of finite sets of natural numbers with list concatenation as arrow composition in C, i.e., $C(\mathtt{Counter}, \mathtt{Counter}) = C(\mathtt{Counter}, d) = (\mathcal{P}_f\omega)^*$.

The behaviour system $[A]$ interprets $\mathtt{Counter}$ as $A_{\mathtt{Counter}}$ and

$$[A]_{[S_1 \dots S_k]:\, \mathtt{Counter} \to \mathtt{Counter}}([S'_1 \dots S'_m]) = [S_1 \dots S_k S'_1 \dots S'_m]$$

$$[A]_{[S_1 \dots S_k]:\, \mathtt{Counter} \to d}([S'_1 \dots S'_m]) = S_1 + \dots + S_k + S'_1 + \dots + S'_m$$

while $[\overline{A}]$ interprets $\mathtt{Counter}$ as $\overline{A}_{\mathtt{Counter}} = D_{\mathtt{Nat}}$ and

$$[\overline{A}]_{[S_1 \dots S_k]}(S) = S_1 + \dots + S_k + S$$

in both cases. This definition can be extended on arrows by using the monotonicity of $+$ wrt the inclusions between finite sets of natural numbers, this making $[\overline{A}]_{[S_1 \dots S_k]}$ functors.

The final behaviour system $\boldsymbol{B}_\Sigma$ interprets $\boldsymbol{B}_{\mathtt{Counter}}$ as $D^{(\mathcal{P}_f\omega)^*}$.

The counter: behaviour image and satisfaction

It is easy to calculate the definition of the component $q_{\mathtt{Counter}}$ of the unique behaviour system morphism $q\colon [A] \to \boldsymbol{B}$:

$$q_{\mathtt{Counter}}([S_1 \dots S_k])([S'_1 \dots S'_m]) = S_1 + \dots + S_k + S'_1 + \dots + S'_m$$

Notice that $q_{\mathtt{Counter}}\colon (\mathcal{P}_f\omega)^* \to D^{(\mathcal{P}_f\omega)^*}$ is *not* full because $A_{\mathtt{Counter}}$ is a discrete category (i.e., a set). It is easy to notice that

$$[A] \to [\overline{A}] \to \boldsymbol{B}$$

is a factorisation for q, therefore $\overline{A}$ is the behaviour image of A. Because of the factorisation system in $\mathbb{C}at$, $\overline{A}_{\mathtt{Counter}} \to \boldsymbol{B}_{\mathtt{Counter}}$ is full, therefore the behaviour image $\overline{A}$ also adds transitions apart of the usual identification between behavioural equivalent states.

Consider the rule

$$[\mathtt{add(M?N, C)}] \Rightarrow [\mathtt{add(M, C)}]$$

It is clear that A *does not* satisfy this rule in the ordinary RWL sense because the model A does not implement any transitions for the counter. However, $A \not\models [\mathtt{add(M?N, C)}] \Rightarrow [\mathtt{add(M, C)}]$ because $\overline{A} \models [\mathtt{add(M?N, C)}] \Rightarrow [\mathtt{add(M, C)}]$.

We can actually prove this rule by using context induction, i.e., proving that

$$\mathtt{COUNTER} \vdash c(\mathtt{add(M?N, C)}) \Rightarrow c(\mathtt{add(M, C)})$$

for each context of visible sort. The only context of length 1 is $\mathtt{read}(z)$, so $\mathtt{read(add(M?N, C))} \Rightarrow \mathtt{read(add(M, C))}$ follows by an application of the structural equation on both sides and by one application of the congruence rule for $\mathtt{M?N} \Rightarrow \mathtt{M}$. Each context c of length $n+1$ is of the form $\mathtt{read}(\mathtt{add}(S, c'))$ where $\mathtt{read}(c'(z))$ is a context of length n. By the structural axiom this reduces to $S + \mathtt{read}(c'(\mathtt{add(M?N, C)})) = \mathtt{read}(c'(\mathtt{add(M, C)}))$ which can be proved by the induction hypothesis and by one application of the rule of congruence.

References

[1] Michael Arbib and Ernest Manes. *Arrows, Structures and Functors.* Academic, 1975.

[2] Francis Borceaux. *Handbook of Categorical Algebra*, volume 2. Cambridge University Press, 1994.

[3] Rod Burstall and Răzvan Diaconescu. Hiding and behaviour: an institutional approach. In A. William Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 75–92. Prentice-Hall, 1994. Also in Technical Report ECS-LFCS-8892-253, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

[4] Răzvan Diaconescu. *Category-based Semantics for Equational and Constraint Logic Programming.* PhD thesis, University of Oxford, 1994.

[5] Răzvan Diaconescu. Completeness of category-based equational deduction. *Mathematical Structures in Computer Science*, 5(1):9–41, 1995.

[6] Răzvan Diaconescu. A category-based equational logic semantics to constraint programming. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification*, volume 1130 of *Lecture Notes in Computer Science*, pages 200–221. Springer, 1996. Proceedings of 11th Workshop on Specification of Abstract Data Types. Oslo, Norway, September 1995.

[7] Răzvan Diaconescu. Category-based modularisation for equational logic programming. *Acta Informatica*, 33(5):477–510, 1996. To appear.

[8] Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularisation. In Gerard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 83–130. Cambridge, 1993. Proceedings of a Workshop held in Edinburgh, Scotland, May 1991.

[9] Kokichi Futatsugi and Toshimi Sawada. Design considerations for cafe specification environment. In *Proc. OBJ2 10th Anniversary Workshop*, October 1995.

[10] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.

[11] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.

[12] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Harmut Ehrig and Fernando Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science*, pages 1–34. Springer, 1994.

[13] Joseph Goguen and Răzvan Diaconescu. An introduction to category-based equational logic. In V.S. Alagar and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 936 of *Lecture Notes in Computer Science*, pages 91–126. Springer, 1995.

[14] Joseph Goguen and Grant Malcolm. A hidden agenda, 1996. draft.

[15] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Algebraic Specification with OBJ: An Introduction with Case Studies.* Cambridge, to appear 1995. Also to appear as Technical Report from SRI International.

[16] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer, 1971.

[17] F. William Lawvere. Functorial semantics of algebraic theories. *Proceedings, National Academy of Sciences, U.S.A.*, 50:869–872, 1963. Summary of Ph.D. Thesis, Columbia University.

[18] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, (93):73–155, 1992.

Electronic Notes in Theoretical Computer Science 4 (1996)

Solving Binary CSP Using Computational Systems

Carlos Castro

INRIA Lorraine & CRIN
615 rue du jardin botanique, BP 101,
54602 Villers-lès-Nancy Cedex, France
e-mail: Carlos.Castro@loria.fr

Abstract

In this paper we formalise CSP solving as an inference process. Based on the notion of Computational Systems we associate actions with rewriting rules and control with strategies that establish the order of application of the inferences. The main contribution of this work is to lead the way to the design of a formalism allowing to better understand constraint solving and to apply in the domain of CSP the knowledge already developed in Automated Deduction.

Keywords: Constraint Satisfaction Problems, Computational Systems, Rewriting Logic.

1 Introduction

In the last twenty years many work has been done on solving Constraint Satisfaction Problems, CSP. The solvers used by constraint solving systems can be seen as encapsulated in black boxes. In this work we formalise CSP solving as an inference process. We are interested in description of constraint solving using rule-based algorithms because of the explicit distinction made in this approach between deduction rules and control. We associate actions with rewriting rules and control with strategies which establish the order of application of the inferences. Our first goal is to improve our understanding of the algorithms developed for solving CSP once they are expressed as rewriting rules coordinated by strategies. Extending the domain of application of Rewriting Logic to CSP is another motivation for this work. This leads the way to the design of a formalism allowing to apply the knowledge already developed in the domain of Automated Deduction. To verify our approach we have implemented a prototype which is currently executable in ELAN [13], a system implementing computational systems.

This paper is organised as follows. Section 2 contains some definitions and notations. Section 3 gives a brief overview of CSP solving. Section 4 presents

© *1996 Elsevier Science B. V.*

in details the computational system we have developed. Finally, Section 5 concludes the paper.

2 Basic Definitions and Notation

In this section we formalise CSP. The basic concepts and definitions that we use are based on [10,11,24].

Definition 2.1 [CSP]

An *elementary constraint* $c^?$ is an atomic formula built on a signature $\Sigma = (\mathcal{F}, \mathcal{P})$, where $\mathcal{F}$ is a set of ranked function symbols and $\mathcal{P}$ a set of ranked predicate symbols, and a denumerable set $\mathcal{X}$ of variable symbols. Elementary constraints can be combined with usual first-order connectives and quantifiers. We denote the set of constraints built from Σ and $\mathcal{X}$ by $\mathcal{C}(\Sigma, \mathcal{X})$. Given a structure $\mathcal{D} = (D, I)$, where D is the carrier and I is the interpretation function, a $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$-CSP is any set $C = \{c_1^?, \ldots, c_n^?\}$ such that $c_i^? \in \mathcal{C}(\Sigma, \mathcal{X})\ \forall i = 1, \ldots, n$.

We denote a set of constraints C either by $C = \{c_1^?, \ldots, c_n^?\}$ or by $C = (c_1^? \wedge \ldots \wedge c_n^?)$. We also denote by $\mathcal{V}ar(c^?)$ the set of free variables in a constraint $c^?$; these are in fact the variables that the constraint constrains. The *arity* of a constraint is defined as the number of free variables which are involved in the constraint:

$$arity(c^?) = \mathcal{C}ard(\mathcal{V}ar(c^?)).$$

In this way we work with a ranked set of constraints $C = \bigcup_{n \geq 0} C_n$, where C_n is the set of all constraints of arity n.

Definition 2.2 [Interpretation]

Let $\mathcal{D} = (D, I)$ be a Σ-structure and $\mathcal{X}$ a set of variables symbols.

- A *variable assignment* wrt I is a function which assign to each variable in $\mathcal{X}$ an element in D. We will denote a variable assignment wrt I by α, and the set of all such functions by $\alpha_{\mathcal{D}}^{\mathcal{X}}$.
- A *term assignment* wrt I is defined as follows:
  - Each variable assignment is given according to α.
  - Each constant assignment is given according to I.
  - If $t_{1\mathcal{D}}, \ldots, t_{n\mathcal{D}}$ are the term assignment of $t_1, \ldots, t_n$ and $f_{\mathcal{D}}$ is the interpretation of the n-ary function symbol f, then $f_{\mathcal{D}}(t_{1\mathcal{D}}, \ldots, t_{n\mathcal{D}}) \in D$ is the term assignment of $f(t_1, \ldots, t_n)$. We will denote a term assignment wrt I and α by $\alpha(t_{\mathcal{D}})$.
- A formula in $\mathcal{D}$ can be given a truth value, true ($\mathbf{T}$) or false ($\mathbf{F}$), as follows:
  - If the formula is an atom $p(t_1, \ldots, t_n)$, then the truth value is obtained by calculating the value of $p_{\mathcal{D}}(t_{1\mathcal{D}}, \ldots, t_{n\mathcal{D}})$, where $p_{\mathcal{D}}$ is the mapping assigned to p by I and $t_{1\mathcal{D}}, \ldots, t_{n\mathcal{D}}$ are the term assignments of $t_1, \ldots, t_n$ wrt I.
  - If the formula has the form $(\neg A), (A \wedge B), (A \vee B), (A \rightarrow B)$, or $(A \leftrightarrow B)$, then the truth value of the formula is given by the following table:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
T	**T**	**F**	**T**	**T**	**T**	**T**
T	**F**	**F**	**F**	**T**	**F**	**F**
F	**T**	**T**	**F**	**T**	**T**	**F**
F	**F**	**T**	**F**	**F**	**T**	**T**

- If the formula has the form $\exists x A$, then the truth value of the formula is true if there exists $d \in D$ such that A has truth value true wrt I and $\alpha_{|x \mapsto d}$, where $\alpha_{|x \mapsto d}$ is α except that x is assigned by d; otherwise, its truth value is false.
- If the formula has the form $\forall x A$, then the truth value of the formula is true if, for all $d \in D$, we have that A has truth value true wrt I and $\alpha_{|x \mapsto d}$; otherwise, its truth value is false.

We denote by $\alpha(A_{\mathcal{D}})$ the interpretation of a formula A wrt I and α.

Definition 2.3 [Satisfiability]

Let Σ be a signature and $\mathcal{D}$ be a Σ-structure:

- Given a formula A and an assignment α, we say that $\mathcal{D}$ *satisfies* A *with* α if
$$\alpha(A_{\mathcal{D}}) = \mathbf{T}.$$
This is also denoted by
$$\mathcal{D} \models \alpha(A).$$
- A formula A is *satisfiable in* $\mathcal{D}$ if there is some assignment α such that
$$\alpha(A_{\mathcal{D}}) = \mathbf{T}.$$
- A is *satisfiable* if there is some $\mathcal{D}$ in which A is satisfiable.

Definition 2.4 [Solution of CSP]

A solution of $c^?$ is a mapping from $\mathcal{X}$ to D that associates to each variable $x \in \mathcal{X}$ an element in D such that $c^?$ is satisfiable in $\mathcal{D}$. The solution set of $c^?$ is given by:
$$Sol_{\mathcal{D}}(c^?) = \{\alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} | \alpha(c^?) = \mathbf{T}\}.$$

A solution of C is a mapping such that all constraints $c^? \in C$ are satisfiable in $\mathcal{D}$. The solution set of C is given by:
$$Sol_{\mathcal{D}}(C) = \{\alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} | \alpha(c_i^?) = \mathbf{T} \ \ \forall i = 1, \ldots, n\}.$$

Finally, in order to carry out the constraint solving process we introduce the following definition:

Definition 2.5 [Membership constraints]

Given a variable $x \in \mathcal{X}$ and a non-empty set $D_x \subseteq D$, the *membership constraint* of x is the relation given by
$$x \in^? D_x.$$
A $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$-CSP C' with *membership constraints* is a $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$-CSP C where $C' = C \bigcup \{x \in ? D_x\}_{x \in \mathcal{X}}$

We use these membership constraints to make explicit the domain reduction process during the constraint solving. In practice, the sets D_x have to be set up to D at the beginning of the constraint solving process, and during the processing of the constraint network they will be eventually reduced. In the standard literature of constraint solving the term *domain reduction* is generally used to make reference to constraint propagation. Since domains are fixed once the interpretation is chosen, the membership constraints allows to propagate the information in a clear and explicit way. ¿From a theoretical point of view, a membership constraint does not differ from a constraint in the set C; its solution set is defined in the same way.

3 Constraint Solving

In this work we consider CSPs in which the carrier of the structure is a finite set and the constraints are only unary or binary. This class of CSP is known as Binary Finite Constraint Satisfaction Problems or simply Binary CSP [17]. For the graphical representation of this kind of CSP general graphs have been used, that is why CSP are also known as networks of constraints [21]. We associate a graph G to a CSP in the following way. G has a node for each variable $x \in \mathcal{X}$. For each variable $x \in \mathcal{V}ar(c^?)$ such that $c^? \in C_1$, G has a loop, an edge which goes from the node associated to x to itself. For each pair of variables $x, y \in \mathcal{V}ar(c^?)$ such that $c^? \in C_2$, G has two opposite directed arcs between the nodes associated to x and y. The constraint associated to arc (x, y) is similar to the constraint associated to arc (y, x) except that its arguments are interchanged. This representation is based on the fact that the first algorithms to process CSP analyse the values of only one variable when they check a constraint.

Example 3.1 Let $\Sigma = (\{3\}, \{\leq, \neq\})$, where $arity(3) = 0, arity(\leq) = arity(\neq) = 2, \mathcal{X} = \{x_1, x_2, x_3\}$, $\mathcal{D} = (\{1, 2, 3, 4, 5\} \subset \mathrm{N}, \{\leq_\mathcal{D}, \neq_\mathcal{D}\})$, and $3_\mathcal{D}, \leq_\mathcal{D}$ and $\neq_\mathcal{D}$ are interpreted as usual in the natural numbers. Considering the $\langle\Sigma, \mathcal{X}, \mathcal{D}\rangle$-CSP $C = \{x_1 \leq^? 3, x_1 \neq^? x_2, x_1 \neq^? x_3, x_2 \neq^? x_3\}$. If we join the membership constraints $x_1 \in^? D_{x_1}, x_2 \in^? D_{x_2}$ and $x_3 \in^? D_{x_3}$ and these sets D_{x_i} are set up to $D_{x_1} = D_{x_2} = D_{x_3} = \{1, 2, 3, 4, 5\}$, the graph which represents this CSP is showed in the Figure 1.

For a given CSP we denote by n the number of variables, by e the number of binary constraints and by a the size of the carrier ($a = \mathcal{C}ard(D)$.) We use $node(G)$ and $arc(G)$ to denote the set of nodes and arcs of graph G, respectively.

Typical tasks defined in connection with CSP are to determine whether a solution exists, and to find one or all the solutions. In this section we present three categories of techniques used in processing CSP: Searching Techniques, Problem Reduction Techniques, and Hybrid Techniques. Kumar's work [14] is an excellent survey on this topic.

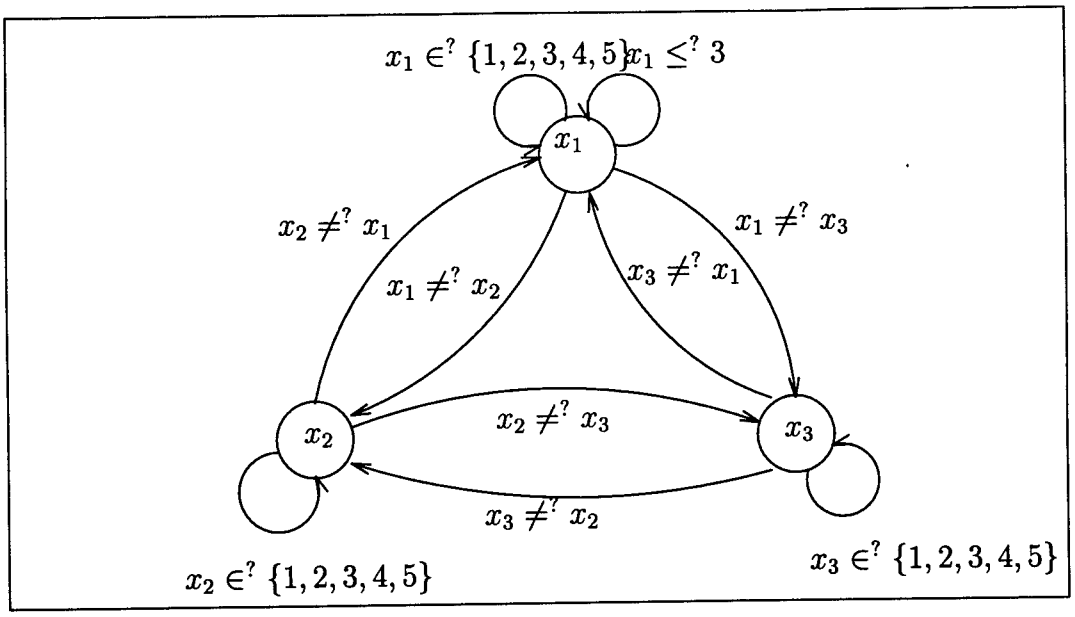


Fig. 1. Graph representation for a Binary CSP

3.1 Searching Techniques in CSP

Searching consists of techniques for systematic exploration of the space of all solutions. The simplest force brute algorithm *generate-and-test*, also called *trial-and-error search*, is based on the idea of testing every possible combination of values to obtain a solution of a CSP. This generate-and-test algorithm is correct but it faces an obvious combinatorial explosion. Intending to avoid that poor performance the basic algorithm commonly used for solving CSPs is the *simple backtracking* search algorithm, also called *standard backtracking* or *depth-first search with chronological backtracking*, which is a general search strategy that has been widely used in problem solving. Although backtracking is much better than generate and test, one almost always can observe pathological behaviour. Bobrow and Raphael have called this class of behaviour *thrashing* [4]. Thrashing can be defined as the repeated exploration of subtrees of the backtrack search tree that differ only in inessential features, such as the assignments to variables irrelevant to the failure of the subtrees. The time complexity of backtracking is $O(a^n e)$, i.e., the time taken to find a solution tends to be exponential in the number of variables [18]. In order to improve the efficiency of this technique, the notion of problem reduction has been developed.

3.2 Problem Reduction in CSP

The time complexity analysis of backtracking algorithm shows that search efficiency could be improved if the possible values that the variables can take is reduced as much as possible [18]. Problem reduction techniques transform a CSP to an equivalent problem by reducing the values that the variables can take. The notion of equivalent problems makes reference to problems which have identical set of solution. Consistency concepts have been defined in order to identify in the search space classes of combinations of values which could

not appear together in any set of values satisfying the set of constraints. Mackworth [17] proposes that these combinations can be eliminated by algorithms which can be viewed as removing inconsistencies in a constraint network representation of the problem and he establishes three levels of consistency: node, arc and path-consistency. These names come from the fact that general graphs have been used to represent this kind of CSP. It is important to realize that the varying forms of consistency algorithms can be seen as *approximation algorithms*, in that they impose *necessary* but not always *sufficient* conditions for the existence of a solution on a CSP.

We now give the standard definitions of node and arc-consistency for a binary network of constraints and we present basic algorithms to achieve them.

3.2.1 Node-Consistency

Definition 3.2 [Node consistency]

Given a variable $x \in \mathcal{X}$ and a unary constraint $c^?(x) \in C$, the node associated to x is *consistent* if

$$\forall \alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} : \alpha \in Sol_{\mathcal{D}}(x \in^? D_x) \Rightarrow \alpha \in Sol_{\mathcal{D}}(c^?(x)).$$

A network of constraints is *node-consistent* if all its nodes are consistent.

Figure 2 presents the algorithm NC-1 which is based on Mackworth [17]. We assume that before applying this algorithm, there is an initialisation step that set up to D the set D_x associated to variable x in the membership constraint $x \in^? D_x$. The time complexity of NC-1 is $O(an)$ [18], so node consistency is always established in time linear in the number of variables by the algorithm NC-1.

```
procedure NC-1;
1  begin
2    for  each x ∈ 𝒳 do
3      for  each α ∈ Sol_𝒟(x ∈? D_x) do
4        if  α(c?(x)) = F then
5          D_x ← D_x\α(x);
6        end_if
7      end_do
8    end_do
9  end
```

Fig. 2. Algorithm NC-1 for node-consistency

3.2.2 Arc-Consistency

Definition 3.3 [Arc consistency]

Given the variables $x_i, x_j \in \mathcal{X}$ and the constraints $c_i^?(x_i), c_j^?(x_j), c_k^?(x_i, x_j) \in C$, the arc associated to $c_k^?(x_i, x_j)$ is *consistent* if

$$\forall \alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} \;\; \exists \alpha' \in \alpha_{\mathcal{D}}^{\mathcal{X}} : \alpha \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i} \wedge c_i^?(x_i))$$

$$\Rightarrow \alpha' \in Sol_{\mathcal{D}}(x_j \in^? D_{x_j} \wedge c_j^?(x_j) \wedge c_k^?(\alpha(x_i), x_j)).$$

A network of constraints is *arc-consistent* if all its arcs are consistent.

The first three algorithms developed to achieve arc-consistency are based on the following basic operation first proposed by Fikes [8]: Given two variables x_i and x_j, both of which are node-consistent, and the constraint $c^?(x_i, x_j)$, if $\alpha \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i})$ and there is no $\alpha' \in Sol_{\mathcal{D}}(x_j \in^? D_{x_j} \wedge c^?(\alpha(x_i), x_j))$ then $\alpha(x_i)$ has to be deleted from D_{x_i}. When that has been done for each $\alpha \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i})$ then arc (x_i, x_j) is consistent (but that no means that arc (x_j, x_i) is consistent.) This idea is embodied in the function REVISE of Figure 3. The time complexity of REVISE is $O(a^2)$, quadratic in the size of the variable's domain [18].

```
function REVISE((x_i, x_j)): boolean
1  begin
2    RETURN ← F ;
3    for each α ∈ Sol_D(x_i ∈? D_{x_i}) do
4      if Sol_D(x_j ∈? D_{x_j} ∧ c?(α(x_i), x_j)) = ∅ then
5        D_{x_i} ← D_{x_i} \ α(x_i);
6        RETURN ← T ;
7      end_if
8    end_do
9  end
```

Fig. 3. Function REVISE

At least one time we have to apply function REVISE to each arc in the graph, but it is obvious that further applications of REVISE to the arcs $(x_j, x_k), \forall x_k \in \mathcal{X}$, could eliminate values in D_{x_j} which are necessary for achieving arc-consistency in the arc (x_i, x_j), so reviewing only once each arc will not be enough. The first three algorithms developed to achieve arc-consistency use the same basic action REVISE but they differ in the strategy they apply REVISE.

Algorithm AC-1

AC-1 reviews, applying REVISE, each arc in an iteration. If at least one set D_x is changed all arcs will be reviewed. This process is repeated until no changes ocurr in all sets. Figure 4 presents the simplest algorithm to achieve arc-consistency, where Q is the set of binary constraints to be reviewed.

The worst case complexity of AC-1 is $O(a^3ne)$ [18]. The obvious inefficiency in AC-1 is that a successful revision of an arc on a particular iteration causes all the arcs to be revised on the next iteration whereas in fact only a small fraction of them could possibly be affected.

Algorithm AC-3

AC-1 can be evidently improved if after the first iteration we only review the arcs which could be affected by the removal of values. This idea was first

```
procedure AC-1;
1  begin
2    Q ← {(x_i, x_j) | (x_i, x_j) ∈ arcs(G), x_i ≠ x_j};
3    repeat
4      change ← false ;
5      for each (x_i, x_j) ∈ Q do
6        change ← change or REVISE((x_i, x_j));
7      end_do
8    until ¬change
9  end
```

Fig. 4. Algorithm AC-1 for arc-consistency

implemented by Waltz' filtering algorithm [26] and captured later by Mackworth's algorithm AC-2 [17]. The algorithm AC-3 proposed by Mackworth [17] also uses this idea. Figure 5 presents AC-3. If we assume that the constraint graph is connected ($e \geq n - 1$) and time complexity of REVISE is $O(a^2)$, time complexity of AC-3 is $O(a^3e)$, so arc-consistency can be verified in linear time in the number of constraints [18].

```
procedure AC-3;
1  begin
2    Q ← {(x_i, x_j) | (x_i, x_j) ∈ arcs(G), x_i ≠ x_j};
3    while Q ≠ ∅ do
4      select and delete any arc (x_i, x_j) ∈ Q;
5      if REVISE((x_i, x_j)) then
6        Q ← Q ∪ {(x_k, x_i) | (x_k, x_i) ∈ arcs(G), x_k ≠ x_i, x_k ≠ x_j};
7      end_if
8    end_do
9  end
```

Fig. 5. Algorithm AC-3 for arc-consistency

In [20] Mohr and Henderson propose the algorithm AC-4 whose worst-case time complexity is $O(ea^2)$ and they prove its optimality in terms of time. AC-4 drawbacks are its average time complexity, which is too near the worst-case time complexity, and even more so, its space complexity which is $O(ea^2)$. In problems with many solutions, where constraints are large and arc-consistency removes few values, AC-3 runs often faster than AC-4 despite its non-optimal time complexity [25]. Moreover, in problems with a large number of values in variable domains and with weak constraints, AC-3 is often used instead of AC-4 because of its space complexity. Two algorithms AC-5 have been developed, one by Deville and Van Hentenryck [7] and another by Perlin [23]. They permit exploitation of specific constraint structures, but reduce to AC-3 or AC-4 in the general case. Bessière [1] proposed the algorithm AC-6 which keeps the optimal worst-case time complexity of AC-4 while working out the drawback of space complexity, AC-6 has an $O(ea)$ space complexity.

However, the main limitation of AC-6 is its theoretical complexity when used in a search procedure. In [2] Bessière proposes an improved version of AC-6, AC-6+, which uses constraint bidirectionality (a constraint is bidirectional if the combination of values a for a variable x_i and b for a variable x_j is allowed by the constraint between x_i and x_j if and only if b for x_j and a for x_i is allowed by the constraint between x_j and x_i.) This algorithm was improved later by Bessière and Régin with their AC-6++ algorithm [3]; by coincidence in the same workshop Freuder presented his AC-7 algorithm [9]. As our aim in this work is to introduce a new framework to model CSP, we use here only AC-1 and AC-3 algorithms because we need a very simple data structures to implement them.

In general, the complexity analysis of consistency algorithms shows that they can be thought of as a low-order polynomial algorithms for exactly solving a relaxed version of a CSP whose solution set contains the set of solutions to the CSP. The more effort one puts into finding the approximation the smaller the discrepancy between the approximating solution set and the exact solution set.

3.3 Hybrid Techniques

As backtracking suffers from thrashing and consistency algorithms can only eliminate local inconsistencies, hybrid techniques have been developed. In this way we obtain a complete algorithm that can solve all problems and where thrashing has been reduced. Hybrid techniques integrate constraint propagation algorithms into backtracking in the following way: whenever a variable is instantiated, a new CSP is created; a constraint propagation algorithm can be applied to remove local inconsistencies of these new CSPs [27]. Embedding consistency techniques inside backtracking algorithms is called Hybrid Techniques. A lot of research has been done on algorithms that essentially fit the previous format. In particular, Nadel [22] empirically compares the performance of the following algorithms: Generate and Test, Simple Backtracking, Forward Checking, Partial Lookahead, Full Lookahead, and Really Full Lookahead. These algorithms primarily differ in the degrees of arc consistency performed at the nodes of the search tree. These experiments indicate that it is better to apply constraint propagation only in a limited form.

4 A Computational System for Solving Binary CSP

The idea of solving constraint systems using computational systems was firstly proposed by Kirchner, Kirchner and Vittek in [12] where they define the concept of computational systems and describe how a constraint solver for symbolic constraints can be viewed as a computational system aimed at computing solved forms for a class of considered formulas called constraints. They point out some advantages of describing constraint solving processes as computational systems over constraint solving systems where solvers are encapsulated in black boxes, such as reaching solved forms more efficiently with smart

choices of rules, easier termination proofs and possibly partly automated, description of constraint handling in a very abstract way, and easy combination of constraint solving with other computational systems. In this section we briefly present computational systems and then describe in details our system for solving Binary CSP.

4.1 Computational Systems

Following [12], a computational system is given by a signature providing the syntax, a set of conditional rewriting rules describing the deduction mechanism, and a strategy to guide application of rewriting rules. Formally, this is the combination of a rewrite theory in rewriting logic [19], together with a notion of strategy to efficiently compute with given rewriting rules. Computation is exactly application of rewriting rules on a term and strategies describe the intented set of computations, or equivalently in rewriting logic, a subset of proofs terms.

4.2 Solved Forms

Term rewriting repeatedly transforms a term into an equivalent one, using a set of rewriting rules, until a normal form is eventually obtained. The solved form we use is defined with the notion of basic form.

Definition 4.1 [Basic form]

A *basic form* for a CSP P is any conjunction of formula of the form

$$\bigwedge_{i\in I}(x_i \in^? D_{xi}) \wedge \bigwedge_{j\in J}(x_j =^? v_j) \wedge \bigwedge_{k\in K}(c^?(x_k)) \wedge \bigwedge_{l,m\in M}(c^?(x_l, x_m))$$

equivalent to P such that

- $\forall i_1, i_2 \in I, i_1 \neq i_2 \Rightarrow x_{i1} \neq x_{i2}$
- $\forall i \in I, D_{xi} \neq \emptyset$
- $\forall j_1, j_2 \in J, j_1 \neq j_2 \Rightarrow x_{j_1} \neq x_{j_2}$
- $\forall i \in I\ \forall j \in J\ x_i \neq x_j$
- $\forall k \in K\ \ \exists i \in I\ \ \exists j \in J, x_k = x_i\ \vee\ x_k = x_j$
- $\forall l \in M\ \ \exists i \in I\ \ \exists j \in J, x_l = x_i\ \vee\ x_l = x_j$

The constraints in the first, second, third and fourth conjunction are called membership, equality, unary and binary constraints, respectively. For each variable we have associated a membership constraint or an equality constraint, the set associated to each variable in the membership constraints must not be empty, and for each variable appearing in the unary or binary constraints there must be associated a membership constraint or an equality constraint. Variables which are only involved in equality constraints are called *solved variables* and the others *non-solved variables.*

A CSP P in basic form can be associated with a *basic assignment* obtained by assigning each variable in the equality constraints to the associated value v and each variable x in the membership constraints to any value in the set D_x.

In this way we can define several forms depending on the level of consistency we are imposing on the constraint set. So, a CSP P in *unary solved form* is a system in basic form whose set of constraints is node consistent, and a CSP P in *binary solved form* is a system in basic form whose set of constraints is arc consistent.

Definition 4.2 [Solved form]

A *solved form* for a CSP P is a conjunction of formulas in basic form equivalent to P and such that all basic assignments satisfy all constraints. A basic assignment of a CSP P in solved form is called *solution.*

4.3 Rewriting Rules

Figure 6 presents **ConstraintSolving**, a set of rewriting rules for constraint solving in CSP. Some ideas expressed in this set of rules are based on Comon, Dincbas, Jouannaud, and Kirchner's work where they present transformation rules for solving general constraints over finite domains [6].

[Node – Consistency]	$x \in^? D_x \ \wedge \ c^?(x) \ \wedge \ C$
	$\Rightarrow \ x \in^? RD(x \in^? D_x, c^?(x)) \ \wedge \ C$
[Arc – Consistency]	$x_i \in^? D_{x_i} \ \wedge \ x_j \in^? D_{x_j} \ \wedge \ c^?(x_i, x_j) \ \wedge \ C$
	$\Rightarrow \ x_i \in^? RD(x_i \in^? D_{x_i}, x_j \in^? D_{x_j}, c^?(x_i, x_j)) \ \wedge$
	$x_j \in^? D_{x_j} \ \wedge \ c^?(x_i, x_j) \ \wedge \ C$
	if $RD(x_i \in^? D_{x_i}, x_j \in^? D_{x_j}, c^?(x_i, x_j)) \neq D_{x_i}$
[Instantiation]	$x \in^? D_x \ \wedge \ C$
	$\Rightarrow \ x =^? \alpha(x) \ \wedge \ C$
	if $\{\alpha\} = Sol_{\mathcal{D}}(x \in^? D_x)$
[Elimination]	$x =^? v \ \wedge \ C$
	$\Rightarrow \ x =^? v \ \wedge \ C\{x \mapsto v\}$
	if $x \in \mathcal{V}ar(C)$
[Falsity]	$x \in^? \emptyset \ \wedge \ C$
	$\Rightarrow \ \mathbf{F}$
[Generate]	$x \in^? D_x \ \wedge \ C$
	$\Rightarrow \ x =^? \alpha(x) \ \wedge \ C$ **or** $x \in^? D_x \backslash \alpha(x) \ \wedge \ C$
	if $\alpha \in Sol_{\mathcal{D}}(x \in^? D_x)$

Fig. 6. **ConstraintSolving**: Rewriting rules for solving Binary CSP

As we explained in section 3.2.2 the first three algorithms to achieve arc-consistency only differ in the strategy they apply a basic action: REVISE.

But, following the main idea of Lee and Leung's Constraint Assimilation Algorithm [15], we can also see the algorithm NC-1 presented in section 3.2.1 as a procedure to coordinate the application of a *domain restriction operation*[1] which removes inconsistent values from the set D_x of the membership constraints. So, we could create only one rewriting rule to implement node and arc-consistency but for clarity reasons we avoid merging both techniques and create the rules **Node-Consistency** and **Arc-Consistency**.

Before applying the algorithm NC-1 we start with the membership constraint $x \in^? D_x$ and the unary constraint $c^?(x)$. After applying NC-1 we obtain a modified membership constraint $x \in^? D'_x$, where D'_x is D_x without the values that satisfy $x \in^? D_x$ but do not satisfy $c^?(x)$. This membership constraint capture all constraint information coming from the original two, their solution sets are the same:

$$Sol_{\mathcal{D}}(x \in^? D'_x) = Sol_{\mathcal{D}}(x \in^? D_x \wedge c^?(x)).$$

This is an inference step where a new constraint can be deduced and the original two be deleted. This key idea is captured by **Node-Consistency**, where $RD(x \in^? D_x, c^?(x))$ stands for the set $D'_x = \{v \in D_x \mid c^?(v)\}$. It is important to note that there is not condition to use this rule because also in case that $c^?(x)$ does not constrain any value already constrained by $x \in^? D_x$ we will not modify the original membership constraint but we can eliminate the constraint $c^?(x)$.

The inference step carried out by arc-consistency algorithms can be seen as an initial state with constraints $x_i \in^? D_{x_i}$, $x_j \in^? D_{x_j}$, and $c^?(x_i, x_j)$ and a final state where $x_i \in^? D_{x_i}$ has been eliminated and a new constraint $x_i \in^? D'_{x_i}$ has been created, where D'_{x_i} corresponds to D_{x_i} without the elements which are not compatible with values in D_{x_j} wrt $c(x_i, x_j)$. This is expressed by the inference rule **Arc-Consistency**, where $RD(x_i \in^? D_{x_i}, x_j \in^? D_{x_j}, c^?(x_i, x_j))$ stands for the set $D'_{x_i} = \{v \in D_{x_i} \mid (\exists w \in D_{x_j})\ \ c^?(v, w)\}$. In this case we require that $RD(x_i \in^? D_{x_i}, x_j \in^? D_{x_j}, c^?(x_i, x_j)) \neq D_{x_i}$ to really go on.

The rewriting rule **Instantiation** corresponds to the variable instantiation. If there is only one assignment α which satisfies $x \in^? D_x$ then the membership constraint is deleted and a new constraint $x =^? \alpha(x)$ is added. This rule makes explicit the dual meaning of an assignment. Algorithmic languages require two different operators for equality and assignment. In a constraint language, equality is used only as a relational operator, equivalent to the corresponding operator in conventional languages. The constraint solving mechanism "assigns" values to variables by finding values for the variables that make the equality relationships true [16].

Elimination express the fact that once a variable has been instantiated we can propagate its value through all constraints where the variable is involved in. In this way we can reduce the arity of these constraints; unary constraints will become ground formulas whose truth value have to be verified and binary constraints will become unary constraints which are more easily tested. Once

[1] This is the name used by Lee and Leung to denote a general operation REVISE which removes inconsistent values of all variables involved in a n-ary constraint $p(x_1, \ldots, x_n)$.

we apply **Elimination** the variable involved in this rule will become a solved variable. It is important to note the strong relation between **Instantiation** and **Elimination**. Semantically the constraints $x \in^? \{v\}$ is equivalent to $x =^? v$, but for efficiency reasons the use of **Elimination** allows the simplification of the constraint system avoiding further resolution of the membership constraint and the constraints where the variable in involved in. Advantages of this approach have been pointed out since the early works on mathematical formula manipulation where the concept of *simplification* was introduced. Caviness [5] mentions that simplified expressions usually require less memory, their processing is faster and simpler, and their functional equivalence are easier to identify. However, it is necessary to point out that with this choice we lose some information, particularly in case of incremental constraint solving, because we do not know any more where the variable was involved in.

The rule **Falsity** express the obvious fact of unsatisfiability. If we arrive to $D_x = \emptyset$ in a membership constraint $x \in^? D_x$ the CSP is unsatisfiable.

The rule **Generate** express the simple fact of branching. Starting with the original constraint set we generate two subsets. In one of them we assume an instantiation for any variable involved in the membership constraints; in the other subset we eliminate that value from the set involved in the membership constraint associated to that variable. In this way the solution for the original problem will be in the union of the solutions for the subproblems.

Lemma 4.3 *The set of rules* **ConstraintSolving** *is correct and complete.*

Proof: Correctness of rule **Node-Consistency** is reduced to prove that $Sol_{\mathcal{D}}(x \in^? RD(x \in^? D_x, c^?(x))) \subseteq Sol_{\mathcal{D}}(\alpha(x \in^? D_x \wedge c^?(x)))$. By definition $RD(x \in^? D_x, c^?(x)) = D'_x$ where $D'_x = \{v \in D_x \mid c^?(v)\}$, so evidently all solution of $x \in^? RD(x \in^? D_x, c^?(x))$ is solution of $x \in^? D_x \wedge c^?(x)$. To prove completeness we can follow the same idea. Correctness and completness of rule **Arc-Consistency** can be proved using the same schema as for **Node-Consistency**. The prove for rules **Instantiation**, **Elimination**, and **Falsity** is evident. The right hand side of rule **Generate** is equivalent to $(x =^? \alpha(x) \vee x \in^? D_x \backslash \alpha(x)) \wedge C$. This expresion is equivalent to $x \in^? D_x \wedge C$, the left hand side of the rule, so rule **Generate** is correct and complete.

Theorem 4.4 *Starting with a CSP P and applying repeatedly the rules in* **ConstraintSolving** *until no rule applies anymore results in* **F** *iff P has no solution or else it results in a solved form of P.*

Proof: Termination of the set of rules is clear since the application of all rules, except one, strictly reduce the size of the set of constraints. The only exception is rule **Instantiation** that does not reduce the set. This rule eliminates a membership constraint and creates an equality constraint. As membership constraint are only created at the beginning of the constraint solving, one for each variable, this rule is applied at most n times.

When we start constraint solving we have the system $C \wedge x \in^? D_x$; $\forall x \in \mathcal{X}$. Rule **Node-Consistency** eliminates unary constraints from C. **Arc-Consistency** only modifies the sets D_x. Rule **Instantiation** eliminates

membership constraints and creates at most one equality constraint per variable. Rule **Eliminate** eventually deletes unary constraints and transforms binary constraints into unary constraints. **Generate** modifies a domain D_x, or deletes a membership constraint and creates an equality constraint. So, if the problem is satisfiable the application of these rules gives a solved form. If the problem is unsatisfiable, i.e., some domain becomes empty, rule **Falsity** will detect that.

4.4 Strategies

As we have mention there are several heuristics to search for a solution in CSP, starting from the brute force generate and test algorithm until elaborated versions of backtracking. The expressive power of computational systems allows to express these different heuristics through the notion of strategy. In this way, for example, a unary solved form can be obtained by applying [**Node-Consistency** | **Falsity**]*, a binary solved form can be obtained by applying [**Arc-Consistency** | **Falsity**]*, and a solved form can be obtained using the strategy [[**Generate; Elimination**] | **Falsity**]* which implements exhaustive searching [2].

We can integrate constraint propagation and searching in order to get a solved form more efficiently than the force brute approach. Let us define the following strategies for applying rules from **ConstraintSolving**:

- **NodeC** :: **Node-Consistency** [[**Instantiation; Elimination**]|**Falsity**]*
- **ArcC** :: **Arc-Consistency** [[**Instantiation; Elimination**]|**Falsity**]*
- **ConsSol1** :: [**NodeC** | **ArcC**]* [[**Generate; Elimination**]|**Falsity**]*
- **ConsSol2** :: [[**NodeC** | **ArcC**]* **Generate; Elimination**]*

The strategy **ConsSol1** implements a preprocessing which verifies node and arc consistency and then carries out an exhaustive search in the reduced problem. The strategy **ConsSol2** implements an heuristic which, once node and arc consistency have been verified, carries out an enumeration step, then verifies again node and arc consistency and so on. **ConsSol2** is a particular version of *Forward Checking* an heuristic widely used in CSP.

4.5 Implementation

We have implemented a prototype of our system which is currently executable in the system ELAN [13], an interpretor of computational systems [3]. To verify our approach we have implemented constraint solving using two versions of arc consistency: AC-1 and AC-3 [4]. The benchmarks which we have carried

[2] The symbol * means applying a given rule zero or N times over the constraint system.

[3] ELAN is available via anonymous ftp at `ftp.loria.fr` in the directory `/pub/loria/protheo/softwares/Elan`. Further information can be obtained at `http://www.loria.fr/equipe/protheo.html/PROJECTS/ELAN/elan.html`

[4] The rewriting system presented in this work allows the direct implementation of AC-1. Implementing AC-3 only required to add a rewriting rule to check the constraints which could be affected by the constraint propagation. For simplicity reasons we do not include

out are consistent with the well known theoretical and experimental results in terms of constraint checking, where AC-3 is obviously better than AC-1. Using the non determinism of ELAN we have easily implemented Forward Checking, the most popular hybrid technique. In Appendix A we present an overview of our implementation. All details about this prototype can be obtained at `http://www.loria.fr/~castro/PROJECTS/csp.h tml`.

5 Conclusion

We have implemented a prototype of a computational system for solving Binary CSP. We have verified how computational systems are an easy and natural way to describe and manipulate Binary CSP. The main contributions of this work can be seen from two points of view. First, we have formalised algorithms to solve Binary CSP in a way which makes explicit difference between actions and control that until now were embeded in black boxes like algorithms. Second, we have extended the domain of application of Rewriting Logic. The distinction between actions and control allows us to better understand the algorithms for constraint solving which we have used. As our aim in this work was only to apply the expressive power of computational systems to better understand constraint propagation in CSP we did not care about efficiency in searching for a solution, so as future work we are interested in efficiency considerations related to our implementation. As a near future work we are interested in the analysis of the data structures which will allow us to implement more efficient versions of arc-consistency algorithms. We hope that powerful strategy languages will allow us to evaluate existing hybrid techniques for constraint solving and design new ones.

Acknowledgement

I am grateful to Dr. Claude Kirchner for his theoretical support and Peter Borovanský and Pierre-Etienne Moreau for their help concerning the implementation.

References

[1] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[2] C. Bessière. A fast algorithm to establish arc-consistency in constraint networks. Technical Report TR-94-003, LIRMM Université de Montpellier II, January 1994.

[3] C. Bessière and J.-C. Régin. An arc-consistency algorithm optimal in the number of constraint checks. In *Proceedings of the Workshop on Constraint Processing, ECAI'94, Amsterdam, The Netherlands*, pages 9–16, 1994.

it here.

[4] D. G. Bobrow and B. Raphael. New Programming Languages for Artificial Intelligence Research. *Computing Surveys*, 6(3):153–174, September 1974.

[5] B. F. Caviness. On Canonical Forms and Simplification. *Journal of the ACM*, 17(2):385–396, April 1970.

[6] H. Comon, M. Dincbas, J.-P. Jouannaud, and C. Kirchner. A Methodological View of Constraint Solving. Working paper, 1996.

[7] Y. Deville and P. V. Hentenryck. An efficient arc consistency algorithm for a class of csp problems. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 325–330, 1991.

[8] R. E. Fikes. REF-ARF: A System for Solving Problems Stated as Procedures. *Artificial Intelligence*, 1:27–120, 1970.

[9] E. C. Freuder. Using metalevel constraint knowledge to reduce constraint checking. In *Proceedings of the Workshop on Constraint Processing, ECAI'94, Amsterdam, The Netherlands*, pages 27–33, 1994.

[10] J. H. Gallier. *Logic for Computer Sciences, Foundations of Automatic Theorem Proving.* Harper and Row, 1986.

[11] H. Kirchner. On the Use of Constraints in Automated Deduction. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 128–146. Springer-Verlag, 1995.

[12] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. V. Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers*, pages 131–158. The MIT press, 1995.

[13] C. Kirchner, H. Kirchner, and M. Vittek. *ELAN, User Manual.* INRIA Loraine & CRIN, Campus scientifique, 615, rue du Jardin Botanique, BP-101, 54602 Villers-lès-Nancy Cedex, France, November 1995.

[14] V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *Artificial Intelligence Magazine*, 13(1):32–44, Spring 1992.

[15] J. H. M. Lee and H. F. Leung. Incremental Querying in the Concurrent CLP Language IFD-Constraint Pandora. In K. M. George, J. H. Carroll, D. Oppenheim, and J. Hightower, editors, *Proceedings of the 11th Anual Symposium on Applied Computing, SAC'96, Philadelphia, Pennsylvania, USA*, pages 387–392, February 1996.

[16] W. Leler. *Constraint Programming Languages, Their Specification and Generation.* Addison-Wesley Publishing Company, 1988.

[17] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[18] A. K. Mackworth and E. C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25:65–74, 1985.

[19] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[20] R. Mohr and T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[21] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[22] B. Nadel. Tree Search and Arc Consistency in Constraint-Satisfaction Algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer-Verlag, 1988.

[23] M. Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.

[24] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT press, 1989.

[25] R. J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc concsistency in CSPs. In *Proceedings IJCAI-93*, pages 239–245, 1993.

[26] D. Waltz. Understanding lines drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[27] M. Zahn and W. Hower. Backtracking along with constraint processing and their time complexities. *Journal of Experimental and Theoretical Artificial Intelligence*, 8:63–74, 1996.

A Implementation

In ELAN, a logic can be expressed by its syntax and its inference rules. The syntax of the logic can be described using mixfix operators. The inference rules of the logic are described by conditional rewrite rules. The language provides three levels of programmation:

- First the design of a logic is done by the so-called *super-user*. In our case that is a description in a generic way of the constraint solving process.
- The logic can be used by the *programmer* in order to write a specification.
- Finally, the *end-user* can evaluate queries valid in the specification, following the semantics described by the logic.

In our implementation the top level of the logic description is given by the *super-user* in the module presented in figure A.1.

This module specifies that the *programmer* has to provide a specification module which has to include two parts: *Variables* and *Values*. As an example we can consider the specification module presented in figure A.2.

The sorts *list, identifier* and *int* are built-in, and the *query sort* and *result sort* are defined by the super-user. Sort list[*formule*] defines the data structure of the query, in this case, a list of constraints. The sort *csp* is a data structure

```
LPL Solver_CSP_Int description
specification description
      part Variables of sort list[identifier]
      part Values of sort list[int]
end
      query of sort list[formule]
      result of sort csp
      modules Solver_CSP[Variables,int,Values]
      start with (Solved_Form) CreateCSP(query)
end of LPL description
```

Fig. A.1. Logic description

```
specification My_variables_and_values
Variables
      X1 X2
Values
      1 2 3 4 5
end of specification
```

Fig. A.2. End-user specification

consisting of three list; the first one records the membership constraints, the second one records the equality constraints, and the third one records the unary and binary constraints. Once the *programmer* has defined the logic, and has provided a query term, ELAN will process in the following way. The symbol *CreateCSP* will apply on the query term, then using the strategy *Solved_Form*, included in the module *Solver_CSP*, ELAN will iterate until no rule applies anymore. *CreateCSP* uses the constructors *CreateLMC*, to create the list of membership constraints, and *CreateC*, to create the list of unary and binary constraints from the list of de formula L given by the *end-user*[5].

The strategy *Solved_Form* control the application of the rules as is showed in the figure A.3. This strategy implements local consistency with exhaustive search. If we eliminate the sub-strategy **dont know choose**(*Generate*) we obtain a particular version of AC-1 algorithm.

Finally, in figure A.4 we present rule *Node-Consistency*. This rule applies the strategy *Strategy_Node-Consistency*, presented in figura A.5, on a *csp* with at least one element in the list of unary and binary constraints. Strategy *Strategy_Node-Consistency* uses rule *GetUnaryConstraint* to get the first unary constraint in the list of unary and binary constraints. If there exists a unary constraint the strategy will apply rule *Node-Consistency_1*, if the variable involved in the unary constraint is in the list of membership constraints, or rule *Node-Consistency_2*, if the variable is in the list of equality constraints. In the set **ConstraintSolving** we use only one rule to verify node consistency, but we have implemented two versions sligtly differents. This is an implementation

[5] Creation of the list of unary and binary constraints is not only a copy of the list L, because for each binary constraint $c^?(x_i, x_j)$ we have to create its inverse $c^?(x_j, x_i)$.

```
strategy Solved_Form
repeat
  dont care choose (
      dont care choose (Node-Consistency)
      ||
      dont care choose (Arc-Consistency)
      ||
      dont care choose (Instantiation)
      ||
      dont care choose (Elimination)
      ||
      dont care choose (Falsity)
      ||
      dont know choose (Generate)
      )
endrepeat
end of strategy
```

Fig. A.3. Strategy *Solved_Form*

choice, as we have a list for the membership constraints and another one for the equality constraints, it is easy to profite this information. The same explanation is valid for arc consistency, where we have created four rules to implement the general version presented in the set **ConstraintSolving**.

```
rules for csp
declare
      x : var;
      v : Type;
      D : list[Type];
      c : formule;
      C,lmc,lec : list[formule];
      P : csp;
bodies
[Node-Consistency] CSP(lmc,lec,c.C) => P
            where P:= (Strategy_Node-Consistency)CSP(lmc, lec,c.C)
end
[Node-Consistency_1] CSP(x in? D.lmc,lec,c.C)
            => CSP(x in? ReviseDxWRTc(x,D,c).lmc,lec,C)
end
[Node-Consistency_2] CSP(lmc,x =? v.lec,c.C) =>
            CSP(lmc,x =? v.lec,C)
            if SatisfyUnaryConstraint(x,v,c)
...
end
```

Fig. A.4. Rules to implement Node Consistency

```
strategy Strategy_Node-Consistency
    dont care choose (GetUnaryConstraint)
    dont care choose (
        dont care choose (GetVarOfUnaryConstraintInLMC)
        dont care choose (Node-Consistency_1)
        ||
        dont care choose (GetVarOfUnaryConstraintInLEC)
        dont care choose (Node-Consistency_2)
        )
end of strategy
```

Fig. A.5. Strategies to implement Node Consistency

Electronic Notes in Theoretical Computer Science 4 (1996)

Bi-rewriting Rewriting Logic*

W. Marco Schorlemmer

Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques
Campus UAB, E-08193 Bellaterra, Catalunya
marco@iiia.csic.es

Abstract

Rewriting logic appears to have good properties as logical framework, and can be useful for the development of programming languages which attempt to integrate various paradigms of declarative programming. In this paper I propose to tend towards the operational semantics for such languages by basing it on *bi-rewrite systems* and *ordered chaining calculi* which apply rewrite techniques to first-order theories with arbitrary possibly non-symmetric transitive relations, because this was an important breakthrough for the automation of deduction in these kind of theories. I show that a proof calculus based on the bi-rewriting technique may serve as framework of different proof calculi, by analizing those of equational logic and Horn logic, and presenting them as specific cases of bi-rewrite systems. Deduction is then essentially bi-rewriting a theory of rewriting logic. Since recently the interest in specifications based on theories with transitive relations has arisen, the result of this research towards a general framework for bi-rewriting based operational semantics of several programming paradigms will also be very useful for the development of rapid prototyping tools for these kind of specifications.

1 Introduction

Term rewriting has been mainly used as a technique for the deduction in equational theories, and was studied thoroughly in the context of rewrite systems [10,40,20]. But recently it has been noticed that, since rewriting is done only in one direction, it is not limited to equivalence relations, but also applicable on arbitrary transitive relations. Indeed, Meseguer showed that the implicit logic underlying rewrite systems is not equational logic, but *rewriting logic* [31]. Meseguer put the strength of his research in developing a strong mathematical semantics of rewriting logic by formulating it as a logic of action and concurrent change.

Similar observation were made independently by Levy and Agustí, as they studied mechanisms for automating the deduction in theories involving sub-

* Supported by project DISCOR (TIC 94-0847-C02-01) funded by the CICYT

© *1996 Elsevier Science B. V.*

set inclusions. They applied rewrite techniques to inclusional theories [25] and generalized the notions of Church-Rosser and termination of rewrite systems to the more general framework called *bi-rewrite systems* [26]. This was an important breakthrough in automated deduction with arbitrary transitive relations: Bachmair and Ganzinger based on Levy and Agustí's work their generalization from *superposition calculi* for full first-order theories with equality [5] to *ordered chaining calculi* for theories with arbitrary transitive relations, besides equality [6]. Actually their calculi apply rewrite techniques (i.e. the use of ordering restrictions on terms and atoms involved in inferences) to the original chaining inference first stated by Slagle [44].

Meseguer's rewriting logic appears to have good properties as logical framework, and, following its approach on 'general logics' [30], different logics of interest have been mapped to it [28]. Therefore a proof calculus for rewriting logic may be useful as framework for a variety of other proof calculi, which can also be mapped to it, specially if such a proof calculus is an effective and, even better, a very efficient one. That's why rewriting logic serves as basis for the development of programming languages like Maude [32], which attempt to unify the paradigms of functional, relational and concurrent object-oriented programming. It was Parker who also advocated programming on non-symmetric transitive relations like preorder or partial order relations for generalizing and subsequently combining several different programming paradigms, symbolic or numeric, like functional and logic programming among others [37,38]. Another recent approach for integrating functional and logic programming, based on rewriting logic, but taking possibly non-deterministic lazy functions as the fundamental notion, has been done by González-Moreno et al. [13].

In order to deal in practice with such multi-paradigm languages like e.g. Maude it is necessary to provide them with an efficient operational semantics. Therefore, instead of formulating it on the straightforward proof calculus defined by the deduction rules of rewriting logic, I argue that by applying the known results about automated deduction in theories with transitive relations, we will be able to define a general framework for the integration of different operational semantics in a more promising way, from the efficiency point of view. In this paper I conjecture that, since the work on bi-rewriting and ordered chaining done by Levy and Agustí, and Bachmair and Ganzinger respectively is suitable for mechanization, their results will be useful for stating such operational semantics framework.

2 Preliminaries

In rewriting logic, a rewrite theory $\mathcal{R}$ can be described as a 4-tuple (F, A, L, R), where (F, A) is a signature consisting of a set F of function symbols and a set A of structural axioms (F-equations like associativity or commutativity), L is a set of labels, and R is a set of sentences of the form $r : [s]_A \Rightarrow [t]_A$ (i.e. labeled rules with $r \in L$) among A-equivalence classes of first-order terms

Reflexivity:

$$\frac{}{[t] \Rightarrow [t]}$$

Congruence: For each $f \in F$,

$$\frac{[s_1] \Rightarrow [t_1] \cdots [s_n] \Rightarrow [t_n]}{[f(s_1, \ldots, s_n)] \Rightarrow [f(t_1, \ldots, t_n)]}$$

Replacement: For each rule $[s] \Rightarrow [t] \in R$,

$$\frac{[u_1] \Rightarrow [v_1] \cdots [u_n] \Rightarrow [v_n]}{[s\langle x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\rangle] \Rightarrow [t\langle x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\rangle]}$$

where $x_1, \ldots, x_n$ are the variables occurring in either s or t.

Transitivity:

$$\frac{[s] \Rightarrow [t] \cdots [t] \Rightarrow [u]}{[s] \Rightarrow [u]}$$

Fig. 1. Deduction rules of rewriting logic

$s, t \in \mathcal{T}(F, X)$ over a denumerable set X of variables[1].

In order to simplify the exposition of the ideas presented in this paper, I will only consider unlabeled rewrite theories, i.e. where rules in R are of the form $[s]_A \Rightarrow [t]_A$. Therefore we can describe such a rewrite theory by means of the triple (F, A, R). When the set of axioms A is clear from the context I will denote the equivalence class of a term t with $[t]$ instead of $[t]_A$.

Given a term expression t, $t|_p$ denotes the subterm occurring at position p. If this occurrence is replaced by term v, we will denote it with $t[v]_p$. A *substitution* $\sigma = \langle x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\rangle$ is a mapping from a finite set $\{x_1, \ldots, x_n\} \subseteq X$ of variables to $\mathcal{T}(F, X)$, extended as a morphism to a mapping from $\mathcal{T}(F, X) \longrightarrow \mathcal{T}(F, X)$. I will use substitutions in postfix notation.

The entailment of sentences $[s] \Rightarrow [t]$ from a rewrite theory $\mathcal{R}$, denoted $\mathcal{R} \vdash_{RWL} [s] \Rightarrow [t]$ is defined by the set of deduction rules given in Figure 1. A rewrite theory $\mathcal{R}$ induces the reachability relation '$\rightarrow_{\mathcal{R}}$', such that $[s] \rightarrow_{\mathcal{R}} [t]$ if we can obtain $[t]$ from $[s]$ by a finite amount of applications of the deduction rules of Figure 1.

An *ordering* $\succ$ is an irreflexive, transitive binary relation. It is a *reduction ordering* if additionally it is well-founded (no infinite sequences of the form $t_1 \succ t_2 \succ \cdots$ exist), monotonic ($u \succ v$ implies $s[u]_p \succ s[v]_p$) and stable under substitutions ($s \succ t$ implies $s\sigma \succ t\sigma$). *Path orderings* define reduction orderings[2] constructing them directly from a well-founded ordering over the symbols of the signature —the *precedence*— by exploring paths in the tree structure of the terms. An example of path ordering is the *lexicographic path*

[1] Actually sentences of rewriting logic are conditional rules [31], but here I will only consider unconditional ones.

[2] Actually they define *simplification orderings* which are reduction orderings satisfying the subterm property $t \succ t|_p$.

ordering. For a complete survey on termination orderings we refer to [9].

3 Proof Calculi for Rewriting Logic

A straightforward proof calculus for rewriting logic is defined by the category with equivalent classes of terms as objects and proof terms as morphisms [28]. Proof terms are built by the deduction rules defining the entailment relation of rewriting logic given in Figure 1 modulo those equations on proof terms, which identify equivalent proofs. Such a proof calculus is based on the following variant of Birkhoff's theorem [8] for the non-symmetric relation '$\Rightarrow$' of rewriting logic:

Lemma 3.1 *Given a rewrite theory $\mathcal{R}$, if '$\rightarrow_{\mathcal{R}}$' denotes the reachability relation induced by the rules of $\mathcal{R}$, then $\mathcal{R} \vdash_{RWL} [s] \Rightarrow [t]$ if and only if $[s] \rightarrow_{\mathcal{R}} [t]$.*

Though for finite theory presentations a decision procedure based on Birkhoff's theorem is implementable (since the set of all theorems of $\mathcal{R}$ is recursively enumerable), it is well known, from equational logic[3], that such a procedure is absolutely intractable and awkward to implement. By first orienting the equations of a theory presentation following a reduction ordering on terms, and subsequently completing such a presentation in order to satisfy the Church-Rosser property, a very efficient proof calculus for equational logic based on normal form computation can be given. But, though normal form computation doesn't have any sense within the more general rewrite theories, we still should consider a proof calculus for rewriting logic which takes such a reduction ordering on terms into account. The fact that sentences of rewriting logic have already an orientation does not imply that such orientation coincides with the direction of term reduction, i.e. $\rightarrow_{\mathcal{R}} \not\subseteq \succ$ in general.

3.1 Bi-rewrite systems

By orienting the sentences $[s] \Rightarrow [t]$ of a given rewrite theory $\mathcal{R} = (F, A, R)$ following an ordering $\succ$ on terms, we obtain *two* separate rewrite relations $\Rightarrow \cap \succ$ and $\Rightarrow \cap \prec$, which I will denote '$\xrightarrow{\Rightarrow}$' and '$\xrightarrow{\Leftarrow}$', respectively, where '$\Rightarrow$' is the direction of the rules in R, and '$\longrightarrow$' is the direction of reduction of terms. We obtain in this way two separate rewrite systems, which form together a *bi-rewrite system* $\langle R_{\Rightarrow}, R_{\Leftarrow} \rangle$.

Example 3.2 *Consider the rewrite theory $\mathcal{R} = (\{a, b, c, f\}, \emptyset, R)$, where R is given below:*

$$R = \begin{cases} f(a, x) \Rightarrow x \\ f(x, c) \Rightarrow x \\ b \Rightarrow f(a, c) \end{cases}$$

[3] As pointed out by Meseguer in [31] equational logic is obtained from rewriting logic by adding the symmetry rule to its deduction rules.

Orienting these rules, following e.g. a lexicographic path ordering based on signature precedence $f \succ c \succ b \succ a$, we obtain the following two rewrite systems $R_{\Rightarrow}$ and $R_{\Leftarrow}$:

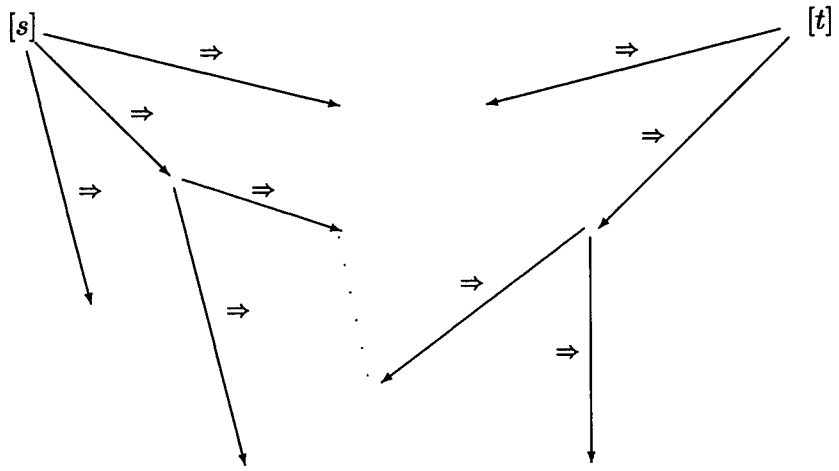
$$R_{\Rightarrow} = \begin{cases} f(a,x) \xrightarrow{\Rightarrow} x \\ f(x,c) \xrightarrow{\Rightarrow} x \end{cases} \qquad R_{\Leftarrow} = \left\{ f(a,c) \xrightarrow{\Leftarrow} b \right.$$

In order to have a decision algorithm for the word problem in a rewrite theory the bi-rewrite system needs to be *convergent*, i.e. it has to satisfy two properties: *Church-Rosser* and *termination*[4]. The system is Church-Rosser if whenever we have two equivalent classes of terms $[s]$ and $[t]$ such that $\mathcal{R} \vdash_{RWL} [s] \Rightarrow [t]$ a *bi-rewrite proof* between these equivalent classes exists, consisting of two paths, one using rules of $R_{\Rightarrow}$ and the other using rules of $R_{\Leftarrow}$, which join together in a common equivalent class:

$$[s] \xrightarrow{\Rightarrow} \cdots \xrightarrow{\Rightarrow} [u] \xleftarrow{\Rightarrow} \cdots \xleftarrow{\Rightarrow} [t]$$

The system is terminating, if no infinite sequences of rewrites with rules in $R_{\Rightarrow}$ (or $R_{\Leftarrow}$) can be built. Termination is guaranteed when the rewrite orderings defined by $R_{\Rightarrow}$ and $R_{\Leftarrow}$ respectively are contained in a unique reduction ordering on terms.

A decision algorithm for the word problem in convergent bi-rewrite systems is then straightforward: To check if $\mathcal{R} \vdash_{RWL} [s] \Rightarrow [t]$ we reduce $[s]$ and $[t]$ applying rewrite rules of each rewrite system, exploring all possible paths, until a common equivalent class of terms is reached:



The conditions put on the rewrite relations in order to guarantee termination also avoid the possibility of infinite branching.

Finite convergent bi-rewrite systems encode the reflexive, transitive and monotone closure of rewrite relation '⇒': All possible consequences of a rewrite theory $\mathcal{R}$ using the deduction rules of rewriting logic can be represented by a bi-rewrite proof.

An arbitrary bi-rewrite system, obtained by orienting the sentences of a rewrite theory $\mathcal{R}$ is non-convergent in general. But, like in the equational case,

[4] To be rigorous we only need quasi-termination [25], but for the sake of simplicity, in this case I require termination.

there exist necessary and sufficient conditions for a terminating bi-rewrite system to be Church-Rosser, which were stated by Levy and Agustí adapting the original results of Knuth and Bendix [21]. First of all we give two definitions and then the theorem which summarizes this result [5]:

Definition 3.3 *Given a bi-rewrite system* $\langle R_{\Rightarrow}, R_{\Leftarrow} \rangle$ *and two rules* $l_1 \stackrel{\Rightarrow}{\longrightarrow} r_1 \in R_{\Rightarrow}$, $l_2 \stackrel{\Leftarrow}{\longrightarrow} r_2 \in R_{\Leftarrow}$ *(or vice versa), and a non-variable subterm* $l_2|_p$, *if* σ *is a most general unifier of* l_1 *and* $l_2|_p$, *then* $\langle l_2[r_1]_p\sigma, r_2\sigma \rangle$ *is called a* critical pair.

Definition 3.4 *Given a bi-rewrite system* $\langle R_{\Rightarrow}, R_{\Leftarrow} \rangle$ *and a rule* $l_1 \stackrel{\Rightarrow}{\longrightarrow} r_1 \in R_{\Rightarrow}$ *and an instance* $l_2\sigma \stackrel{\Leftarrow}{\longrightarrow} r_2\sigma$ *of a rewrite rule* $l_2 \stackrel{\Leftarrow}{\longrightarrow} r_2 \in R_{\Leftarrow}$ *(or vice versa), where* σ *is such that, for some term* v *with subterm* $v|_q = l_1$, *and some variable* x *at position* p *that appears more than once in* l_2, $x\sigma = v$ *and* $y\sigma = y$, *whenever* $y \neq x$, *then the critical pair* $\langle l_2[v[r_1]_q]_p\sigma, r_2\sigma \rangle$ *is called a* variable instance pair [6].

A critical or variable instance pair is said to be *convergent* if it has a bi-rewrite proof, and *divergent* otherwise.

Theorem 3.5 (Levy and Agustí [25]) *A terminating bi-rewrite system* $\langle R_{\Rightarrow}, R_{\Leftarrow} \rangle$ *is Church-Rosser (and thus, convergent) if and only if there are no divergent critical or variable instance pairs between the rules of* $R_{\Rightarrow}$ *and the rules of* $R_{\Leftarrow}$.

Following the same ideas proposed by Knuth and Bendix, one can attempt to complete a non-convergent terminating bi-rewrite system, by means of adding divergent critical and variable instance pairs as new rewrite rules to the systems $R_{\Rightarrow}$ or $R_{\Leftarrow}$. Notice that the number of critical pairs among rewrite rules of sets $R_{\Rightarrow}$ and $R_{\Leftarrow}$ is always finite. But from the definition of variable instance pairs, we can observe that the overlap of term l_1 on l_2 is done *below* a variable position of l_2, and therefore unification always succeeds. Furthermore, term v is arbitrary, which means that if a variable instance pair exists between two rewrite rules then there are an infinite number of them. As we will see later, this is one of the major drawbacks for the tractability of the generalization of rewrite techniques to arbitrary transitive relations, because a completion procedure which attempts to add variable instance pairs as new rewrite rules is impossible to manage in general.

We know from the completion of equational theories, that the process may fail to orient a critical pair with the given reduction ordering. There have been various variants of completion to overcome this situation [22,39,18], which have been also generalized to bi-rewrite systems [26].

[5] For the sake of simplicity I present Levy and Agustí's results for the case where no structural axioms are considered, i.e. $A = \emptyset$.

[6] Variable instance pairs also appear in the context of rewriting modulo a congruence [2].

3.2 Ordered chaining

During the last decade and the beginning of the present it has been shown that the process of completion of rewrite systems can be seen as a process of refutation in the context of resolution-based theorem proving [15,3]. The principle of refutation by means of resolution is the core of the operational semantics of the logic programming paradigm [27].

Completion as a refutation process was later generalized for full first-order theories with equality [14,5] and has been further improved [35,7]. This generalization is also applicable to completion of bi-rewrite systems, and consequently we can prove theorems of a theory in rewriting logic applying a process of refutation captured by the *ordered chaining calculus* of Bachmair and Ganzinger [6]. It is based on the ordered chaining inference rule between two clauses and in essence generalizes the critical pair and variable instance pair computation during completion of bi-rewrite systems. The inference rule is stated as follows:

$$\textbf{Ordered Chaining:}\quad \frac{C \vee s \Rightarrow t \quad D \vee u \Rightarrow v}{C\sigma \vee D\sigma \vee u[s]_p\sigma \Rightarrow v\sigma}$$

where σ is a most general unifier of t and $u|_p$, p being a subterm position in u, and the following ordering restrictions between terms, and literals hold: $s\sigma \not\succeq t\sigma$, $v\sigma \not\succeq u\sigma$, $s\sigma \Rightarrow t\sigma$ is the strictly maximal literal with respect to the remaining disjunction $C\sigma$ of the first clause, and $u\sigma \Rightarrow v\sigma$ is the strictly maximal literal with respect to the remaining disjunction $D\sigma$ of the second clause. In this context, as in the equational case, the process of completion, is known as *saturation.*

The complete calculus for full first-order clauses with transitive relations is formed of the ordered chaining inference rule together with several other inference rules —negative chaining, ordered resolution, ordered factoring and transitivity resolution—, which also put ordering restriction on the terms and atoms participating in the inference, in order to prune the search space to be explored (see [6] for further details). Bachmair and Ganzinger proved the refutational completeness of the calculus by means of their 'model construction method': Given a *saturated set*[7] of clauses they inductively construct —over an ordering on clauses— a Herbrand interpretation which is the minimal model of the saturated set. This model is then a preordered set. They also gave an intuitive notion of *redundant* clauses and inferences within the context of this model construction method. This notion is very important, since in analogy to a completion procedure, which attempts to produce a convergent bi-rewrite system in which all critical pairs and variable instance pairs are convergent (have a bi-rewrite proof), the saturation process attempts to provide us a set of clauses in which all inferences are redundant. We say in this case that the set of clauses is *saturated*, i.e. closed up to redundancy. Notice that this is the criterion in order to finish the process of completion, or saturation respectively. In the same manner as during the completion process rewrite rules are kept as interreduced as possible, during saturation redundant

[7] I give the meaning of saturated set below.

clauses are deleted, and redundant inferences avoided, by means of so called *redundancy provers*. Unfortunately, unlike the equational case, there is a lack of powerful redundancy proving techniques that can be used within a theorem prover dealing with arbitrary transitive relations.

3.3 Drawbacks of the general ordered chaining calculus

We have seen that calculi based on bi-rewriting, like ordered chaining, are suitable as proof calculi for rewriting logic, since ordering restriction on terms and atoms significantly prune the search space of the prover. But these calculi are still highly prolific in the general case [42]. Inferences require unification on variable positions, although only when they appear repeated in the same term (see Definition 3.4), and, if the operators are monotonic with respect to the transitive relation (e.g. the rewrite relation '$\Rightarrow$' in rewrite theories) functional reflexive axioms are explicitly needed, in order to make variable instance pairs convergent. On the other hand, no rewriting within equivalence classes of terms is done, making a notion of unique normal form, on which equational term rewriting is based, meaningless. Consequently the order of application of rewrite rules is now significant, making term rewriting shift from *don't care* nondeterminism to *don't know* nondeterminism: Backtracking is needed for a rewrite proof to be found[8]. But by restricting these calculi to special theories, or by limiting the kind of axioms we use, it is possible to provide rewriting logic with interesting subcalculi. It is known, e.g. that in dense total orderings without endpoints, variable chaining can be avoided completely [4]. Furthermore completion of the inclusional theory of lattices to a finite and convergent bi-rewrite system is possible [24] (though no finite term rewrite system for the equational theory of lattices exists [11]) and this fact suggests to consider the properties of specific algebraic structures for improving deduction in rewriting logic.

4 A Framework for Proof Calculi

In this section I present the idea that a proof calculus based on the bi-rewriting technique may serve as framework of different proof calculi. I will sketch this on two very intuitive and well-known logics, following Martí-Oliet and Meseguer's approach in [28], mapping them to rewriting logic.

I am going to present the proof calculi of equational logic and Horn clause logic, from the perspective of bi-rewriting. This may appear strange or even absurd in a first sight, but my purpose is to show that these operational semantics are in fact specific cases of bi-rewrite system, and that their special nature restrict significantly the general proof calculus based on bi-rewriting. Furthermore, these restrictions act upon the drawbacks I just mentioned in Section 3.3.

[8] In spite of these general drawbacks, there exists an implementation in Prolog of a theorem prover based on ordered chaining, done by Nivela, Nieuwenhuis and Ganzinger, called *Saturate* [36,12], for which currently better implementation techniques are studied [34].

4.1 Bi-rewriting equational logic

An equational theory $\mathcal{E}$ can be described as a triple (F, A, E), where (F, A) is a signature consisting of a set F of function symbols and a set A of structural axioms (F-equations), and E is a set of equations of the form $[s]_A = [t]_A$ between equivalence classes of terms. Note that if A is the empty set, the equations in E are between terms.

An equational theory $\mathcal{E} = (F, A, E)$ is mapped to a rewrite theory $\mathcal{R} = (F, A, R)$, such that for every equation $[s] = [t]$ in E, two rules $[s] \Rightarrow [t]$ and $[t] \Rightarrow [s]$ are in R, in order to make explicit the property of symmetry. The bi-rewrite system $\langle R_{\Rightarrow}, R_{\Leftarrow} \rangle$ resulting from orienting the rules of R has for every rule $[s] \xrightarrow{\Rightarrow} [t]$ in $R_{\Rightarrow}$ also a rule $[s] \xrightarrow{\Leftarrow} [t]$ in $R_{\Leftarrow}$, i.e. each former equation appears as a rewrite rule in both rewrite systems.

Example 4.1 *Let's consider the map of equational theory $\mathcal{E} = (\{+, s, 0\}, \emptyset, E)$ —which specifies the non-associative/commutative sum operator— into rewrite theory $\mathcal{R} = (\{+, s, 0\}, \emptyset, R)$ given below:*

$$E = \begin{cases} x + 0 = x \\ x + s(y) = s(x + y) \end{cases} \longmapsto R = \begin{cases} x + 0 & \Rightarrow x \\ x & \Rightarrow x + 0 \\ x + s(y) & \Rightarrow s(x + y) \\ s(x + y) & \Rightarrow x + s(y) \end{cases}$$

Orienting the rules in R, following e.g. a lexicographic path ordering based on the signature precedence $+ \succ s \succ 0$, we get the following bi-rewrite system:

$$R_{\Rightarrow} = \begin{cases} x + 0 & \xrightarrow{\Rightarrow} x \\ x + s(y) & \xrightarrow{\Rightarrow} s(x + y) \end{cases} \qquad R_{\Leftarrow} = \begin{cases} x + 0 & \xrightarrow{\Leftarrow} x \\ x + s(y) & \xrightarrow{\Leftarrow} s(x + y) \end{cases}$$

Due to symmetry, we actually are duplicating each rewrite rule. Note that since generation of critical pairs is done by looking for overlaps between left-hand sides of two rules, one of each rewrite system, in this case this is equivalent to look for overlaps among the rules of one unique rewrite system, i.e. rules that actually rewrite on equations. When dealing with equational theories, bi-rewrite systems can be 'simplified' to standard rewrite systems, as we are familiar, as for instance the following equational term rewrite system for the equational theory of Example 4.1:

$$R' = \begin{cases} x + 0 & \xrightarrow{=} x \\ x + s(y) & \xrightarrow{=} s(x + y) \end{cases}$$

Such rewrite systems correspond e.g. to the semantics of Maude's functional modules [32].

Overlaps on variable positions and the functional reflexive axioms are not needed: All those overlaps are convergent, because rewrite rules appear in both rewrite systems (see [42]). If the set of equations E is Church-Rosser (in the 'traditional' sense of equational rewrite systems, for instance see [10]), the bi-rewrite system $\langle R_{\Rightarrow}, R_{\Leftarrow} \rangle$ obtained from set of rules R in which E is

mapped to is also Church-Rosser (in the sense of Theorem 3.5), as well as each of both rewrite systems $R_{\Rightarrow}$ and $R_{\Leftarrow}$ (again in the equational sense).

In the case A is not empty, rewriting must be done modulo the set of axioms in A. As mentioned in Section 3.1 this has been thoroughly studied by the rewriting community, and their results can be applied also to bi-rewrite systems. This suggests that Patrick Viry's notion of *coherence completion* [45] for the implementation of rewriting in rewriting logic by using standard rewriting instead of rewriting modulo, should be also applicable to bi-rewrite systems.

Symmetry plays an important role, because when reasoning with equivalence relations, we can deal with the notion of *equivalence class*. Since we do not have two different rewrite systems any more, critical pairs are computed by overlapping left-hand sides of rules of one unique rewrite system. If such rewrite system is convergent this has important practical consequences: Each term not only has an irreducible term, the so called *normal form*, but this normal form is also unique for each term. Rewriting is done within an equivalence class, and all the members of this class share the same normal form. A decision procedure for the word problem in equational theories, based on convergent rewriting systems, is much simpler than in arbitrary rewrite theories. Just the normal forms of the two terms of the equation we want to validate are computed and checked for identity. Furthermore the property of *don't care* nondeterminism of theorem proving in convergent equational theories is kept.

4.2 Bi-rewriting Horn logic

A Horn theory $\mathcal{H}$ can be described as a 4-tuple (F, P, A, H). The triple (F, P, A) is the signature, consisting of a set F of function symbols, a set P of predicate symbols, and a set A of structural axioms (i.e. F-equations). H is a set of Horn clauses of the form $[s]_A \longleftarrow [t_1]_A, \ldots, [t_n]_A$. A Horn theory $\mathcal{H} = (F, P, A, H)$ is mapped to a two-sorted rewrite theory $\mathcal{R} = (F \cup P', A \cup A', R)$ with sorts **term** and **prop**. All functions symbols in F take arguments of sort **term** and are themselves of sort **term**, and set P' contains a constant *true* of sort **prop**, a binary infix operator '$\wedge$' of sort **prop** taking as argument two elements of sort **prop**, and for each n-ary predicate p in P, an n-ary function symbol p of sort **prop** taking as arguments n elements of sort **term**. A' is the set containing the associativity, commutativity and identity law (with respect to constant *true*) of operator '$\wedge$', and R is the set of rules obtained by mapping each clause $[s]_A \longleftarrow [t_1]_A, \ldots, [t_n]_A$ to the rule $[s]_{A \cup A'} \Rightarrow [t_1 \wedge \cdots \wedge t_n]_{A \cup A'}$, and each unit clause $[s]_A$ to the rule $[s]_{A \cup A'} \Rightarrow [true]_{A \cup A'}$.

Example 4.2 *Horn theory* $\mathcal{H} = (\{ann, bob, tom\}, \{par, anc\}, \emptyset, H)$ *—which specifies the parent (par) and ancestor (anc) relation— is mapped to rewrite theory* $\mathcal{R} = (\{ann, bob, tom, par, anc, true, \wedge\}, A', R)$ *as follows,* A' *being the set defined above:*

$$H = \begin{cases} par(ann, bob) \\ par(bob, tom) \\ anc(x,y) \leftarrow par(x,y) \\ anc(x,y) \leftarrow par(x,z), anc(z,y) \end{cases} \mapsto R = \begin{cases} par(ann, bob) \Rightarrow true \\ par(bob, tom) \Rightarrow true \\ anc(x,y) \Rightarrow par(x,y) \\ anc(x,y) \Rightarrow par(x,z) \wedge anc(z,y) \end{cases}$$

4.2.1 SLD-resolution is not bi-rewriting

It is well-known that a proof calculus based on the resolution inference is efficient as operational semantics for Horn logic programming: Queries to a program are existentially quantified formulas $\exists \bar{x}\ u_1, \ldots, u_m$ [9], and are solved by refuting its negation. A resolution step is then as follows [10]:

$$\frac{\longleftarrow u_1, u_2, \ldots, u_m \qquad s \longleftarrow t_1, \ldots, t_n}{\longleftarrow t_1\sigma, \ldots, t_n\sigma, u_2\sigma, \ldots, u_m\sigma}$$

where σ is a most general unifier of u_1 and s.

A query in its correspondent rewrite theory reads then $\exists \bar{x}\ [u_1 \wedge u_2 \wedge \cdots \wedge u_m]_{A'} \Rightarrow [true]_{A'}$, which is solved also by refuting its negation. The inference step which corresponds to the above resolution step reads:

$$\frac{[u_1 \wedge u_2 \wedge \cdots \wedge u_m] \not\Rightarrow [true] \qquad [s] \Rightarrow [t_1 \wedge \cdots \wedge t_n]}{[t_1\sigma \wedge \cdots \wedge t_n\sigma \wedge u_2\sigma \wedge \cdots u_m\sigma] \not\Rightarrow [true]} \quad (1)$$

where σ is, as before, a most general unifier of u_1 and s. This inference step is actually a *negative chaining* (see [6]).

Since chaining is only done through the term on the left-hand side of the rule representing the negated query, until a term in the A'-equivalence class of *true* is reached, we can see this inference also as applying rule $[s] \Rightarrow [t_1 \wedge \cdots \wedge t_n]$ in order to *narrow* [11] the 'query term' $[u_1 \wedge u_2 \wedge \cdots \wedge u_m]$:

$$[u_1 \wedge u_2 \wedge \cdots \wedge u_m] \rightsquigarrow [t_1\sigma \wedge \cdots \wedge t_n\sigma \wedge u_2\sigma \wedge \cdots u_m\sigma]$$

Here σ is, again, a most general unifier of u_1 and s. This is the approach followed by C. Kirchner, H. Kirchner and Vittek in [19], who also studied the map of proofs in Horn theories to proofs in rewrite theories. They map Horn clauses to narrowing rules, and the proof-theoretic structure of Horn logic, based on SLD-resolution, is therefore captured by the straightforward application of the deduction rules of rewriting logic. They further add to the rewrite theory a notion of strategy to efficiently compute with the given rewrite rules and call such a rewrite theory plus strategy a *computational system.*

Negative chaining —Inference 1 above— is ordered if rules $[s] \Rightarrow [t_1 \wedge \cdots \wedge t_n]$ of rewrite theory $\mathcal{R}$ are oriented from left to right, i.e. $[s] \succ [t_1 \wedge \cdots \wedge t_n]$. Indeed, the operational behavior of query solving in Horn theories following resolution strategies known from logic programming, like Prolog's SLD-resolution, is captured by the trivial 'bi-rewrite' system $(R_{\Rightarrow}, \emptyset)$, where $\Rightarrow \subseteq \xrightarrow{\Rightarrow}$. This 'bi-

[9] $\bar{x}$ denotes the free variables of terms $u_1, \ldots, u_m$.

[10] For the sake of simplicity this inference is shown for Horn theories with no structural axioms, i.e. $A = \emptyset$.

[11] Narrowing was originally devised as an efficient E-unification procedure using convergent sets of rewrite rules [16].

rewrite' system is actually a standard rewrite system since we are not rewriting in two directions, and its operational behavior corresponds to standard deduction in rewriting logic. But, as said in Section 3 the ordering induced by these rules will not be in general a reduction ordering, and therefore this 'bi-rewrite' system will in general be non-terminating.

4.2.2 Ordered chaining for Horn theories

When taking a reduction ordering on terms into account, the process of theorem proving in Horn logic maps to an ordered chaining inference tree. I will show this through an example.

Example 4.3 *If we orient the rules of the rewrite theory obtained in Example 4.2 following e.g. a lexicographic path ordering based on the signature precedence* $\wedge \succ anc \succ par \succ tom \succ bob \succ ann \succ true$*, we get the following bi-rewrite system:*

$$R_{\Rightarrow} = \begin{cases} par(ann, bob) \xrightarrow{\Rightarrow} true \\ par(bob, tom) \xrightarrow{\Rightarrow} true \\ anc(x, y) \xrightarrow{\Rightarrow} par(x, y) \end{cases}$$

$$R_{\Leftarrow} = \left\{ par(x, z) \wedge anc(z, y) \xrightarrow{\Leftarrow} anc(x, y) \right.$$

As said in Section 3.1, by orienting the rules of a rewrite theory by means of a reduction ordering on terms, critical pairs (or even variable instance pairs) among the rules of both rewrite systems can arise: We need to start a process of completion for proving theorems, by generating new rules, i.e. our proof calculus will be based on ordered chaining (see Section 3.2). The interesting point is that, since the unique operator of the signature which is monotonic with respect to the relation '$\Rightarrow$' is the the conjunction operator '$\wedge$', the overlap required for generating new rules is only needed on whole propositions and not on terms within them. Furthermore, since the map of Horn to rewrite theories does not introduce variables as arguments of '$\wedge$', unification on variable positions is not needed, and the intractable variable instance pair generation can be completely avoided (and therefore functional reflexive axioms are superfluous).

Figure 2 shows the ordered chaining inference tree for proving theorem $anc(ann, tom) \Rightarrow true$ in rewrite theory $\mathcal{R}$ of Example 4.2. The leaf with the framed sentence is the negation of the theorem. All other leafs are sentences of the rewrite theory. Inference steps are labeled with *(OC)* if it is a ordered chaining step, with *(NC)* if it is a negative chaining step and with *(OR)* if it is a ordered resolution step (see [6] for further details). Bold faced terms are the ones who are unified (i.e. chained through). For instance the top most inference step of Figure 2 corresponds to the generation of a critical pair among rewrite rules $par(x, z) \wedge anc(z, y) \xrightarrow{\Leftarrow} anc(x, y)$ and $anc(x', y') \xrightarrow{\Rightarrow} par(x', y')$.

$$\dfrac{\dfrac{\dfrac{anc(x,y) \Rightarrow par(x,z) \wedge \mathbf{anc(z,y)} \qquad \mathbf{anc(x',y')} \Rightarrow par(x',y')}{anc(x,y) \Rightarrow \mathbf{par(x,z)} \wedge par(z,y) \qquad \mathbf{par(ann,bob)} \Rightarrow true}\ (OC)}{\boxed{\mathbf{anc(ann,tom)} \not\Rightarrow true} \qquad \begin{array}{c} anc(ann,y) \Rightarrow true \wedge par(bob,y) \\ \| \| \| \ A' \\ \mathbf{anc(ann,y)} \Rightarrow par(bob,y) \end{array}}\ (NC)}{par(bob,tom) \not\Rightarrow true \qquad par(bob,tom) \Rightarrow true}\ (OR)$$

□

Fig. 2. Ordered chaining inference tree

Unfortunately, as we can observe from Figure 2, the linear strategy of resolution in Horn theories must be —for completeness— abandoned, since the generation of rules from critical pairs (i.e. ordered chaining inference steps) correspond to resolution among clauses of the given theory. But the advantages of the use of term ordering arise, when it is possible to saturate (i.e. to complete) a bi-rewrite system obtained from the previously explained map: The search for proofs by SLD-resolution (or straightforward deduction in rewriting logic, see Section 4.2.1), which could have been non-terminating, is now 'replaced' by terminating bi-rewriting (because of the reduction ordering on terms).

Example 4.4 *Given the following set of Horn clauses:*

$$q(x) \longleftarrow p(x)$$
$$p(x) \longleftarrow q(x)$$

and the (negated) query:

$$\longleftarrow q(a)$$

Though it is evident that we cannot refute it, the process of applying SLD-resolution will never terminate. Instead, given a signature precedence $q \succ p$, the rewrite theory to which this Horn theory is mapped, forms a convergent bi-rewrite system:

$$R_{\Rightarrow} = \{q(x) \xrightarrow{\Rightarrow} p(x)\}$$
$$R_{\Leftarrow} = \{p(x) \xrightarrow{\Leftarrow} q(x)\}$$

Now we can proof, in a finite amount of time, that $q(a) \not\Rightarrow true$, because $q(a) \xrightarrow{\Rightarrow} p(a)$ is the only rewrite step that can be performed.

Further work I want to do in this direction is to study the results about termination of Horn clause programs from this point of view, and to reformu-

late the conditions of termination as restrictions on proof calculi of rewriting logic.

5 Towards a Framework for the Operational Semantics of Logic Programs

Martí-Oliet and Meseguer conjecture in [29], that rewriting logic can be useful as logical framework, at least for those logics we can consider of 'practical interest', and whose proof calculi correspond to the operational semantics of programming languages based on these logics. In this paper I have made a first step towards the study of specific restrictions on bi-rewriting based calculi by analizing mappings between proof calculi, which I think will be useful for defining a general notion of operational semantics: Research in this direction will be promising.

Furthermore, recently the interest in specifications based on logics with transitive relations has arisen. Mosses introduced *unified algebras* [33], a framework for the algebraic specification of abstract data types, where sorts are treated as values, so that operations may be applied to sorts as well as to the elements that they classify. This framework is based on a partial order of a distributive lattice with a bottom. Similar intuitions were followed by Levy and Agustí, who proposed the *Calculus of Refinements* [24], a formal specification model based on inclusions. Their approach showed to be useful for the preliminary specification and further stepwise refinement of complex systems [41]. Rewriting logic itself and its embodiment in Maude has served as prototyping language for the specification of complex systems [23]. Therefore the result of this research towards the design of a multi-paradigm programming language dealing with arbitrary transitive relations may also be very useful for developing rapid prototyping tools for these kind of specifications [43].

Besides these general specification frameworks, partial orders also play a central role in a variety of much more concrete logic programming languages. For example, Aït-Kaci and Podelski make use of order-sorted feature terms as basic data structure of the programming language LIFE [1], generalizing in this way the flat first-order terms normally used as unique data structure in logic programming. An order-sorted feature term is a compact way to represent the collection of elements of a given non-empty domain which satisfy the constraint encoded by the term, and therefore may be interpreted itself as a sort, like in 'unified algebras' or in the 'Calculus of Refinements', being LIFE one of the first proposals of sorts as values. Algebraically, a term denotes an element of a meet semi-lattice with a top $\top$ and a bottom $\bot$, which in essence is a subalgebra of the power set of the considered domain. But, deduction in LIFE is quite poor, because of the restricted use of terms within the definition of the partial order. Deduction reduces to unification of order-sorted feature terms and can be seen as the meet operation in the semi-lattice. It is performed by normalizing the conjunction of the constraints encoded in the terms to be unified, and is equivalent to intersecting the collections of elements the terms represent.

Also Jayaraman, Osorio and Moon base their *partial order programming* paradigm on a lattice structure, and are specially interested on the complete lattice of finite sets [17]. In their paradigm they pursue the aim to integrate sets into logic programming, and to consider them as basic data structure on which the paradigm relies. But in this framework no deduction mechanisms are given to validate order related functional expressions.

To summarize, in a future work it is necessary to analize proof calculi and theorem proving strategies of different interesting logics, and to study the map of their proof calculi to bi-rewriting. This will clarify how efficiency issues and strategies of these calculi are captured by restrictions on general calculi based on bi-rewriting, so that a sufficiently general proof calculus of rewriting logic based on bi-rewriting and ordered chaining can be stated, which may serve as general framework for the operational semantics of interesting logic programming and specification paradigms I have just mentioned. The knowledge about these restrictions translated to efficiency aspects of proof calculi will help to find an optimal balance between generality and efficiency.

Acknowledgement

I am specially grateful to Jaume Agustí for his valuable comments and helpful suggestions on previous versions of this paper.

References

[1] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16:195–234, 1993.

[2] L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Symposium of Logic in Computer Science*, pages 346–357, 1986.

[3] L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion without failure. In *Resolution of Equations in Algebraic Structures*, volume 2. Academic Press, 1989.

[4] L. Bachmair and H. Ganzinger. Ordered chaining for total orderings. In A. Bundy, editor, *Automated Deduction — CADE'12*, volume 814 of *LNAI*, pages 435–450. Springer-Verlag, 1994.

[5] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):1–31, 1994.

[6] L. Bachmair and H. Ganzinger. Rewrite techniques for transitive relations. In *Proc., Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 384–393, 1994.

[7] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In D. Kapur, editor, *Automated Deduction — CADE-11*, volume 607 of *LNAI*, pages 462–476. Springer-Verlag, 1992.

[8] G. Birkhoff. On the structure of abstract algebras. *Proc. Cambridge Philos. Soc.*, 31:433–454, 1935.

[9] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[10] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. Elsevier Science Publishers, 1990.

[11] R. Freese, J. Ježek, and J. Nation. Term rewrite systems for lattice theory. *Journal of Symbolic Computation*, 16:279–288, 1993.

[12] H. Ganzinger, R. Nieuwenhuis, and P. Nivela. The Saturate system. `http://www.mpi-sb.mpg.de/SATURATE/Saturate.html`, 1995.

[13] J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In H. R. Nielson, editor, *Programming Languages and Systems — ESOP '96*, LNCS 1058. Springer-Verlag, 1996.

[14] J. Hsiang and M. Rusinowitch. Proving refutational completeness of theorem proving strategies: The transfinite semantic tree method. *Journal of the ACM*, 38(3):559–587, 1991.

[15] G. Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *Journal of Computation and System Sciences*, 23:11–21, 1981.

[16] J. M. Hullot. Canonical forms and unification. In *Proc. 4th International Conference on Automated Deduction*, LNCS 87, 1980.

[17] B. Jayaraman, M. Osorio, and K. Moon. Partial order programming (revisited). In *Proc. Algebraic Methodology and Software Technology (AMAST)*, pages 561–575, 1995.

[18] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15:1155–1194, 1986.

[19] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. van Hentenryck and S. Saraswat, editors, *Principles and Practice of Constraint Programming.* MIT Press, 1995.

[20] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[21] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[22] D. S. Lankford and A. Ballantyne. Decision procedures for simple equational theories with permutative axioms: Complete sets of permutative reductions. Technical Report ATP-37, Department of Mathematics and Computer Science, University of Texas, 1977.

[23] U. Lechner, C. Lengauer, and M. Wirsing. An object-oriented airport: Specification and refinement in Maude. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Types Specification*, LNCS 906. Springer Verlag, 1995.

[24] J. Levy. *The Calculus of Refinements: a Formal Specification Model Based on Inclusions.* PhD thesis, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, 1994.

[25] J. Levy and J. Agustí. Bi-rewriting, a term rewriting technique for monotonic order relations. In C. Kirchner, editor, *Rewriting Techniques and Applications*, LNCS 690, pages 17–31. Springer-Verlag, 1993.

[26] J. Levy and J. Agustí. Bi-rewrite systems. *Journal of Symbolic Computation*, 1996. To be published.

[27] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[28] N. Martí-Oliet and J. Meseguer. Rewriting logic as logical and semantic framework. Technical Report SRI-CSL-93-05, Computer Science Laboratory, SRI International, August 1993.

[29] N. Martí-Oliet and J. Meseguer. General logics and logical frameworks. In D. M. Gabbay, editor, *What is a Logical System?*, pages 355–391. Clarendon Press, 1994.

[30] J. Meseguer. General logics. In H. D. Ebbinghaus et al., editors, *Logic Colloquium '87*, pages 275–329. Elsevier Science Publishers, 1989.

[31] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Journal of Theoretical Computer Science*, 96:73–155, 1992.

[32] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha et al., editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 315–390. MIT Press, 1993.

[33] P. Mosses. Unified algebras and institutions. In *Principles of Programming Languages Conference*, pages 304–312. ACM Press, 1989.

[34] R. Nieuwenhuis, J. M. Rivero, and M. A. Vallejo. An implementation kernel for theorem proving with equality clauses. In *Proc. of the 1996 Joint Conference on Declarative Programming APPIA-GULP-PRODE'96*, pages 89–103, July 1996.

[35] R. Nieuwenhuis and A. Rubio. Basic superposition is complete. In *European Symposium on Programming*, 1992.

[36] P. Nivela and R. Nieuwenhuis. Saturation of first-order (constrained) clauses with the Saturate system. In C. Kirchner, editor, *Rewriting Techniques and Applications*, LNCS 690, pages 436–440. Springer-Verlag, 1993.

[37] D. S. Parker. Partial order programming. Unpublished monograph, 1987.

[38] D. S. Parker. Partial order programming. In *POPL'89: 16th ACM Symposium on Principles of Programming Languages*, pages 260–266. ACM Press, 1989.

[39] G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, 1981.

[40] D. A. Plaisted. Equational reasoning and term rewriting systems. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, pages 273–364. Oxford University Press, 1993.

[41] D. Robertson, J. Agustí, J. Hesketh, and J. Levy. Expressing program requirements using refinement lattices. *Fundamenta Informaticae*, 21:163–183, 1994.

[42] W. M. Schorlemmer and J. Agustí. Theorem proving with transitive relations from a practical point of view. Research Report IIIA 95/12, Institut d'Investigació en Intel·ligència Artificial (CSIC), May 1995.

[43] W. M. Schorlemmer and J. Agustí. Inclusional theories in declarative programming. In *Proc. of the 1996 Joint Conference on Declarative Programming APPIA-GULP-PRODE'96*, pages 167–178, July 1996.

[44] J. R. Slagle. Automated theorem proving for theories with built-in theories including equality, partial orderings and sets. *Journal of the ACM*, 19:120–135, 1972.

[45] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE '94, Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe*, LNCS 817, pages 648–660. Springer-Verlag, July 1994.

Electronic Notes in Theoretical Computer Science 4 (1996)

Specifying Real-Time Systems in Rewriting Logic

Peter Csaba Ölveczky [a,b,1] José Meseguer [a,2]

[a] *Computer Science Laboratory*
SRI International
Menlo Park, U. S. A.

[b] *Department of Informatics*
University of Bergen
Bergen, Norway

Abstract

This work investigates the suitability of rewriting logic as a semantic framework for modeling real-time and hybrid systems. We present a general method to specify and symbolically simulate such systems in rewriting logic and illustrate it with a well-known benchmark. We also show how a wide range of real-time and hybrid system models can be naturally expressed and are unified within our approach. The relationships with timed rewriting logic [9,10] are also investigated.

1 Introduction

Rewriting logic is a flexible and expressive framework in which many different models of concurrent computation and many different types of systems can be naturally specified [13,16,12,14]. It seems therefore natural to investigate the question of how rewriting logic can be applied to the specification of real-time and hybrid systems. From the semantic point of view this offers the possibility of integrating real-time aspects with other features and models already supported by rewriting logic.

The first important research contribution exploring the application of rewriting logic to real-time specification has been the work of Kosiuczenko and Wirsing on *timed rewriting logic* (TRL) [9], an extension of rewriting logic where the rewrite relation is labeled with time stamps. TRL has been shown well-suited for giving object-oriented specifications of complex hybrid systems such

[1] Supported by the Norwegian Research Council.

[2] Supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, National Science Foundation Grant CCR-9224005, and by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

© *1996 Elsevier Science B. V.*

as the steam-boiler [18]. In fact, rewriting logic object-oriented specifications in the Maude language [16] have a natural extension to TRL object-oriented specifications in *Timed Maude* [9,18].

The approach taken here is different. We do not extend rewriting logic at all. We instead investigate the question of how naturally, and with what degree of generality, can real-time systems be formally specified in standard rewriting logic. Our findings indicate that real-time and hybrid systems can in fact be specified quite naturally in rewriting logic, and that a wide range of frequently-used models of real-time and hybrid systems can be regarded as special instances of our theoretical approach.

In essence our approach can be summarized as follows. A time domain—satisfying quite general axioms, so as to allow both discrete and continuous, as well as linear and nonlinear time models—is an explicit parameter of the specification. In addition, the passage of time is viewed as a monoid action, acting on the states, whose result on different components of the system is specified by rewrite rules. The system can then *react* to such time actions and to stimuli from its environment, by performing control actions that are also specified by rewrite rules. In some cases such reaction rules must have an eager strategy, to ensure that real-time requirements are met.

In Section 2 we explain the general method for specifying real-time systems sketched above. We then illustrate in Section 3 the naturalness of the method, and its smooth integration with rewriting logic's support for object-oriented specification, by means of the well-known train intersection controller benchmark. The question of how generally rewriting logic can be used to express other real-time and hybrid system models is addressed in Section 4. We show in detail how a wide range of such models, including timed automata [3], hybrid automata [2], timed and phase transition systems [11], and timed extensions of Petri nets [1,17], can indeed be expressed in rewriting logic quite naturally and directly.

In Section 5 we study the relationships between our approach and TRL. We show that there is a map of entailment systems $\mathcal{M} : TRL \longrightarrow RWL$ sending each TRL specification to a corresponding specification in such a way that logical entailment is preserved. However, the translated theory $\mathcal{M}(\mathcal{T})$ can in general prove additional sentences. This is due to some intrinsic conceptual differences between both formalisms that our analysis reveals. However, for the cases of TRL theories $\mathcal{T}$ where $\mathcal{M}(\mathcal{T})$ exactly mirrors the deductions of $\mathcal{T}$ we provide a general method not only for performing corresponding deductions, but also for simulating through execution in rewriting logic the behavior of the system specified by $\mathcal{T}$.

Indeed, *symbolic simulation* of a real-time system's formal specification is one of the attractive features of our general approach. Perhaps a good way to see how rewriting logic specifications complement more abstract specifications such as temporal logic as well as more concrete, automaton-based ones, is to think of them as providing an *intermediate level*, that can substantially help in bridging the gap between specification and implementation by providing:

- a precise mathematical model of the system (the initial model [13]) against

which more abstract specifications can be proved correct;

- support for useful kinds of automated or semi-automated reasoning about the system at the rewriting logic and equational logic levels;
- support for executable specification and symbolic simulation;
- good system compositionality capabilities, through parameterization, module hierarchies, and object-oriented features.

Much more should be done to further investigate and exploit these possibilities. The formal tools already available, and those planned for the near future, will greatly help us and others in this regard.

2 Specifying Real-Time Systems in Rewriting Logic

This section introduces the rewriting logic techniques we use for specifying and reasoning about two different aspects of real-time systems. Section 2.1 gives abstract specifications of time. In Section 2.2 the idea of time as an action, acting on a system so as to change its state, is introduced, obtaining a framework where properties of the form "t rewrites to t' in time r" can be proved. In Section 2.3 these ideas are extended and used to simulate the actual behavior of systems. In this second case, we are interested in whether a term t, representing a given state, rewrites to a term t' in arbitrary time. The evolution in time of a system can then be observed by following the rewrite sequence starting with the initial state t. In Section 2.4 some rewriting strategies that should sometimes be included in a specification are briefly outlined.

Notation: We will use the symbols $r, r', r_1, \ldots$ to denote time *values*, $x_r, y_r, \ldots$ to denote *variables* of sort *Time*, and $t_r, t'_r, \ldots$ to denote *terms* of sort *Time*.

2.1 Time Models

Time is modeled abstractly by a commutative monoid $(Time, +, 0)$ with additional operators $\leq$, $<$, and $\dot{-}$ ("monus") satisfying the following Maude theory.

fth *TIME* **is**
protecting *BOOL*
  sort *Time*
  op $0 : \rightarrow Time$
  op $_+_ : Time, Time \ [assoc\ comm\ id : 0]$
  ops $_<_, _\leq_ : Time, Time \rightarrow Bool$
  op $_\dot{-}_ : Time, Time \rightarrow Time$
  vars $x_r, y_r, z_r, w_r : Time$
  ceq $x_r = 0$ **if** $(x_r + y_r) == 0$
  ceq $y_r = z_r$ **if** $x_r + y_r == x_r + z_r$
  eq $(x_r + y_r) \dot{-} y_r = x_r$
  ceq $x_r \dot{-} y_r = 0$ **if** $not(y_r \leq x_r)$

eq $x_r \leq x_r + y_r = true$
ceq $(x_r \leq y_r) = true$ **if** $(x_r < y_r)$
eq $(x_r < x_r) = false$
eq $(x_r \leq y_r) = (x_r < y_r)$ or $(x_r == y_r)$
ceq $x_r + y_r \leq z_r + w_r$ **if** $x_r \leq z_r$ and $y_r \leq w_r$
ceq $(x_r \dot{-} y_r) + y_r = x_r$ **if** $y_r \leq x_r$
endft

In this theory, it can be proved that the relation $_ \leq _$ is a partial order, that for all $x_r, y_r : Time$, $0 \leq x_r = true$, and that $y_r \leq x_r$ if and only if there exists a unique z_r (namely $x_r \dot{-} y_r$) such that $x_r = y_r + z_r$.

For simulation and executable specification purposes we will be interested in *computable* models of the above theory *TIME*. This means that all the operations are computable. By the Bergstra-Tucker Theorem [5], such models are finitely specifiable as initial algebras for a set E of Church-Rosser and terminating equations. For example, the nonnegative rational numbers can be so specified as a model of *TIME* by adding a subsort Rat_+ to the specification of rationals in [7], and extending it with an order and a monus operation in the obvious way. Similarly, the real algebraic numbers with the standard order are also computable [19], and therefore, have a finite algebraic specification with Church-Rosser and terminating equations. Note that just taking a constructive version of the real numbers will not yield a computable data type, because the equality and order predicates on the constructive reals are not computable [4].

In many cases, an additional time value ∞ is needed.

fth $TIME_\infty$ **is**
extending *TIME*
sort $Time_\infty$
subsort $Time \leq Time_\infty$
op $\infty : \rightarrow Time_\infty$
op $_ \leq _ : Time_\infty, Time_\infty \rightarrow Bool$
ops $_ + _, _ \dot{-} _ : Time_\infty, Time_\infty \rightarrow Time_\infty$ $[assoc\ comm\ id : 0]$
var $x_r : Time$
eq $x_r \leq \infty = true$
endft

In case time is assumed to be linear, such as in the railroad crossing example in Section 3, time can be specified by the following theory:

fth *LTIME* **is**
extending *TIME*
op min : $Time, Time \rightarrow Time$ $[comm]$
vars $x_r, y_r : Time$
ceq $x_r = y_r$ **if** $not(x_r < y_r)$ and $not(y_r < x_r)$
ceq $\min(x_r, y_r) = y_r$ **if** $y_r \leq x_r$
endft

This theory can also be extended with an ∞ time value as follows:

fth $LTIME_\infty$ **is**
extending $LTIME$, $TIME_\infty$
  op $\min : Time_\infty, Time_\infty \to Time_\infty$ $[comm]$
  var $x_r : Time_\infty$
  eq $\min(\infty, x_r) = x_r$
endft

2.2 Time as an Action

Our proposal is to view the passage of time as an *action* that has an effect on each state of the system. As we have just explained, time is modeled as a commutative monoid $(Time, +, 0)$ with additional structure. Therefore, the action in question is axiomatized as a monoid action δ satisfying the usual axioms:

$$\delta(y, 0) = y$$
$$\delta(y, x_r + x'_r) = \delta(\delta(y, x_r), x'_r)$$

where y is a variable of any sort corresponding to the system's state.

In addition, other rewrite rules describe how a state on which time acts is transformed into an ordinary state. Intuitively, the meaning of δ is that for $t, t' \in T_{\Sigma-\{\delta\}}$, the sequent

$$\delta(t, r) \longrightarrow t'$$

is a valid rewrite deduction in the theory iff it is the case that whenever time has acted on t for r time units, it could rewrite to t'. The following simple example shows how rewriting logic can thus be used to deduce temporal properties of a system.

Example 2.1 Assume that time is modeled by the natural numbers, that if time acts on a term a for two time units, a can rewrite to b, which can then rewrite to c if time has acted for time 0, and, finally, c can rewrite to d in two time units. This system can be modeled by the rewriting logic theory with constants a, b, c, d of sort $State$, an operation $\delta : State, Nat \to State$, the above monoid action equations, and the set of rules

$$\{\delta(a, 2) \longrightarrow b,\ b \longrightarrow c,\ \delta(c, 2) \longrightarrow d\}.$$

It is easy to deduce that $\delta(a, 4) \longrightarrow d$, which means that when time has acted on a for 4 time units, it could rewrite to d.

2.3 Simulation and "Ticks"

Although the time monoid action itself is in some ways sufficient to reason about time change, in many cases we are interested in *simulating* in rewriting logic the behavior of a real-time system in terms of the ordinary states of which it is made up. Therefore, instead of just proving time elapse properties of the form $\delta(t, r) \longrightarrow t'$, we wish to start with a term t representing a given state and then simulate the system, i.e., observe its evolution in time in the

form of sequences of rewrites

$$t \longrightarrow t' \longrightarrow t'' \longrightarrow \cdots .$$

Some care must be taken in this case, since the understanding in the previous section was that, for t, t' not involving the δ operator, $t \longrightarrow t'$ is supposed to mean that t rewrites to t' in zero time ($t = \delta(t, 0) \longrightarrow t'$). We need to carefully change the intended meaning of the rewrite relation to let $t \longrightarrow t'$ denote, under appropriate assumptions, that t could rewrite to t' in arbitrary time.

Another problem stems from the fact that a simulation of this kind clearly requires "tick" rewrites, which model the elapse of time in a system. Using arbitrary rules of the form

$$t \longrightarrow \delta(t, t_r)$$

presents the risk of allowing rewrites such as $f(t, t') \longrightarrow f(\delta(t, t_r), t')$, i.e., rewrites in which time elapses only in a part of the system under consideration.

The solution to both of these problems is to assume that the states of the system are in a sort *State*, and then introduce a new sort *GlobalState* and a new symbol $\widehat{\ } : \mathit{State} \rightarrow \mathit{GlobalState}$ whose intended meaning is that the state t of the *whole* system under consideration is denoted by $\widehat{t}$. Therefore, $\widehat{t} \longrightarrow \widehat{t'}$ means that the whole system in state t rewrites to the state t' in some time, while

$$\delta(t, r) \longrightarrow t' \text{ and } t \longrightarrow t'$$

still mean that (the part) t (of a system) rewrites to t' in time r, and in no time, respectively.

Tick rules modeling the elapse of time therefore have the operator $\widehat{\ }$ at the top, to ensure uniform time elapse. Furthermore, since time may not elapse for certain deadlock states, the tick rules are in general conditional, and hence of the form

$$\widehat{t} \longrightarrow \widehat{\delta(t, t_r)} \text{ if } C.$$

Hence, a real-time system specified by a rewrite theory $\mathcal{R}$ with rules of the form $\delta(t, r) \longrightarrow t'$ (including instantaneous rules where $r = 0$) can be further specified for simulation purposes by means of a rewrite theory $\widehat{\mathcal{R}}$, where $\widehat{\mathcal{R}}$ contains $\mathcal{R}$ plus the following additional data:

(i) a new sort *GlobalState* and an operator $\widehat{\ } : \mathit{State} \rightarrow \mathit{GlobalState}$, and

(ii) tick rules of the form $\widehat{t} \longrightarrow \widehat{\delta(t, t_r)}$ if C.

Then, for any such $\widehat{\mathcal{R}}$ and terms t and t' not containing the symbol δ, a deduction $\widehat{\mathcal{R}} \vdash \widehat{t} \longrightarrow \widehat{t'}$ implies that there is an $r : \mathit{Time}$ such that $\mathcal{R} \vdash \delta(t, r) \longrightarrow t'$, and that then we can find a proof $\alpha : \widehat{t} \longrightarrow \widehat{t'}$ involving exactly n applications of "tick" rules, each advancing the time by $r_1, \ldots, r_n$, respectively, so that the additive equation $r = r_1 + \cdots + r_n$ holds. Therefore, in a simulation of this nature, we can not only observe the different evolutions in time of a system, but we can also measure the elapsed time by inspecting the rewrite proof corresponding to a given evolution.

Furthermore, if we only add the unconditional tick rules $\widehat{y} \longrightarrow \widehat{\delta(y, x_r)}$ for $y : State, x_r : Time$, we have $\widehat{\mathcal{R}} \vdash \widehat{t} \longrightarrow \widehat{t'}$ iff there is a time $r : Time$ such that $\mathcal{R} \vdash \delta(t, r) \longrightarrow t'$.

Example 2.2 (Example 2.1 cont.) Assuming that time only proceeds from states a and c, the system in Example 2.1 could be extended either by the tick rules $\{\widehat{a} \longrightarrow \widehat{\delta(a, 2)}, \widehat{c} \longrightarrow \widehat{\delta(c, 2)}\}$ or by the most general tick rules $\widehat{y} \longrightarrow \widehat{\delta(y, x_r)}$. In either case, we have $\forall t, t' \in T_{\Sigma - \{\delta\}}$, $\widehat{\mathcal{R}} \vdash \widehat{t} \longrightarrow \widehat{t'}$ iff $\delta(t, r) \longrightarrow t'$ for some $r \in \{0, 2, 4\}$.

In some cases, we could relax the general method by having tick rules of the form $\widehat{t} \longrightarrow \widehat{t'}$ **if** C which combine the effect of a tick and a "δ-rule". For instance, in Example 2.2, we could have tick rules $\widehat{a} \longrightarrow \widehat{b}$ and $\widehat{c} \longrightarrow \widehat{d}$ "directly" instead of the tick rules $\widehat{a} \longrightarrow \widehat{\delta(a, 2)}$ and $\widehat{c} \longrightarrow \widehat{\delta(c, 2)}$. While saving some rewrite steps and avoiding the symbol δ, this method has the disadvantage that it is not possible to extract information about the elapsed time from a proof $\alpha : \widehat{t} \longrightarrow \widehat{t'}$.

2.4 Simulation and Strategies

A real-time system runs, as it were, a "race against time". It often has to meet deadlines and to ensure that appropriate actions happen in a timely fashion. That is, among the different transitions that can be taken at a particular point in time, some may have top priority.

¿From a rewriting logic point of view this means that the specification of a real-time system may include not only the rewrite rules specifying its possible transitions, but also a *rewriting strategy*, which further constrains the correct rewrite behavior of the system. Using the reflective features of rewriting logic this can be done in a fully declarative way using a strategy language that is *internal*, and whose semantics is given by rewrite rules [6].

Some specifications do not need any strategies. When a strategy is needed, it has the following very simple form: the set $\mathcal{R}$ of rewrite rules is divided into a set $\mathcal{R}_{eager}$ of *eager* rules and a set $\mathcal{R}_{lazy}$ of *lazy* rules. Intuitively, an eager rule should be applied whenever enabled. Therefore, the rewriting strategy imposed by this division is that

no rule in $\mathcal{R}_{lazy}$ may be applied if any rule in $\mathcal{R}_{eager}$ is enabled.

If no eager rule is enabled, a one-step concurrent $\mathcal{R}_{lazy}$-rewrite [13] may take place. In case $\mathcal{R}_{eager}$ is empty, rules can be applied with no restrictions. In this paper, eager rules will be indicated by the keyword **eager**.

In the rewriting logic framework suggested above for real-time systems, the set $\mathcal{R}$ of rules can be split into sets $\mathcal{R}_{time}$ and $\mathcal{R}_{inst}$, where $\mathcal{R}_{time}$ lets time elapse in a system, and $\mathcal{R}_{inst}$ defines the instantaneous state changes. Furthermore, $\mathcal{R}_{time}$ can often be divided into two sets $\mathcal{R}_{tick}$ and $\mathcal{R}_{\delta}$, where $\mathcal{R}_{tick}$ lets time act on a system and $\mathcal{R}_{\delta}$ defines *how* time acts on it. Some of the state changes modeled by instantaneous rules are required to take place as soon as possible, that is, before time elapses. Therefore, the rules in $\mathcal{R}_{tick}$

are lazy, while $\mathcal{R}_{inst}$ may contain both eager and lazy rules.

Furthermore, tick rules should not advance the time beyond a point at which an eager instantaneous state change could have taken place. In this paper, this is ensured by appropriate conditions on the tick rules, and is not part of the rewriting strategy. However, for sufficiently complex systems it may be more convenient to preclude advancing time too much not by conditions on tick rules, but instead by a more sophisticated strategy than eagerness.

Although this is not a semantic requirement, for simulation purposes it can be quite convenient to apply the rules in $\mathcal{R}_\delta$ with an eager strategy. This has the advantage of being able to determine at what time an instantaneous rule was applied by inspection of a sequence of rewrites. However, by using the exchange rule stating equivalence of rewrite proofs [13], if the rules in R_δ are applied with a different strategy, it is always possible to normalize the proof so that such times can still be determined.

3 Example: Railroad Crossing in Rewriting Logic

In this section we show how the described framework for specifying real-time systems in rewriting logic can be used to give a Maude specification of a railroad crossing controller.

3.1 The Problem

The generalized railroad crossing (see, e.g. [8]) is a benchmark example of real-time systems. The system operates a gate at a railroad crossing. The crossing I lies in a region of interest R. A set of trains travel through R on multiple tracks. A sensor system determines when each train enters the region R (so that the gate can be down when, later on, the train enters the intersection I), and when it exits the intersection I.

The control program reacts to these enter and exit messages by sending messages for raising and lowering a gate to the environment. The system must satisfy that, whenever there is a possibility that a train is in the intersection, the gate should be down. However, the gate should be up as much as possible. We assume that:

- more than one train can be in R on the same track at the same time, and
- the minimum time for a train to enter I after entering R is R_to_I, the time to lower a gate (either from a raising position or when the gate is up) is denoted *time_lower*. Hence, the (minimum) time ϵ from the time a train enters R until the gate must be lowered (in case it is not down or lowering already) is given by $\epsilon = R_to_I - time_lower \geq 0$.

The solution should consist of the following parts:

(i) a specification of the control program, and

(ii) a model of the environment, which is used for simulation and validation purposes

satisfying the above requirements.

3.2 Outline of the Solution

The structure of the solution is straightforward: a *Xing* object represents the state of the control program, and an *env* object provides a simplistic model of the environment. The total system is the composition of these two objects.

Since for the purpose of controlling the crossing it does not matter on which tracks the trains are located, in the crossing object trains are represented by a multiset of time values, where a value r indicates that there is a train in the region, and that it could reach the intersection in time r; a value 0 indicates that a train could be in I.

The crossing object also includes the gate status. State *down* indicates that the gate is lowering or is down. Otherwise it is *up*.

The $TIME_\infty$ theory is extended to multisets of time as follows:

fmod $MULTI\text{-}TIME[T :: LTIME_\infty]$ **is**
  sort $Multi_time$
  subsort $Time \leq Multi_time$
  op $\emptyset_t : \rightarrow Multi_time$
  op $_\ _ : Multi_time, Multi_time \rightarrow Multi_time\ [assoc\ com\ id : \emptyset_t]$
  op $least : Multi_time \rightarrow Time_\infty$
  op $_ \dot{-} _ : Multi_time, Time \rightarrow Multi_time$
  vars $x_r, y_r : Time$
  var $m : Multi_time$
  eq $least(\emptyset_t) = \infty$
  eq $least(x_r\ m) = \min(x_r, least(m))$
  eq $\emptyset_t \dot{-} x_r = \emptyset_t$
  eq $(y_r\ m) \dot{-} x_r = (y_r \dot{-} x_r)\ (m \dot{-} x_r)$
endfm

The whole system, which we assume consists of only one crossing and one environment object, is given as a parameterized object-oriented module $XING[T :: TIME_\infty]$ with the following declarations:

protecting $MULTI\text{-}TIME[T]$
  sorts $Gatestate, GlobalConfiguration$
  ops $up, down : \rightarrow Gatestate$
  op $\delta : Configuration, Time \rightarrow Configuration$
  op $\widehat{\ } : Configuration \rightarrow GlobalConfiguration$
  ops $R_to_I, time_lower, time_raise, time_car : \rightarrow Time$
  vars $x_r, y_r : Time$
  vars $x, e : Oid$
  vars $m, m' : Multi_time$
  class $Xing \mid trains : Multi_time, gstate : Gatestate$
  msgs $enterR, exitI : \rightarrow Msg$

The *enterR* message is handled as follows:

$$\textbf{eager}\ (enterR)\langle x : Xing | trains : m\rangle \longrightarrow \langle x : Xing | trains : m\ R_to_I\rangle.$$

When a train exits I, the gate could be raised if there is time for a car to pass after the gate is up, and before the gate needs to be lowered again, otherwise it stays down:

$$\begin{aligned}&\textbf{eager } (exitI)\langle x:Xing|trains:0\ m, gstate:down\rangle \longrightarrow\\ &\qquad \textbf{if } least(m) - time_lower \geq time_car + time_raise\\ &\qquad \textbf{then } \langle x:Xing|trains:m, gstate:up\rangle(raise)\\ &\qquad \textbf{else } \langle x:Xing|trains:m, gstate:down\rangle.\end{aligned}$$

Whenever the gate is up (or raising) and some train could have reached a time at which the gate should be lowered, it is lowered:

$$\begin{aligned}&\textbf{eager } \langle x:Xing|trains:time_lower\ \ m, gstate:up\rangle \longrightarrow\\ &\qquad \langle x:Xing|trains:time_lower\ \ m, gstate:down\rangle(lower).\end{aligned}$$

Time acts on a *Xing* object in the following way:

$$\textbf{eager } \delta(\langle x:Xing|trains:m\rangle, x_r) \longrightarrow \langle x:Xing|trains:m \dot{-} x_r\rangle.$$

3.3 The Environment

The behavior of the environment is quite simple. It consumes *lower* and *raise* messages that cause the appropriate actions; it produces *enterR* messages at any time, and it can produce *exitI* messages within certain time constraints relative to *enterR* messages. Here we assume that the time from the instant a train reaches R until it exits I is between ϵ_1 and ϵ_2 where $\epsilon_2 > \epsilon_1$ and $\epsilon_1 > R_to_I$.

In the environment object the set of trains is represented as a multiset of times, where for each train we keep the amount of time that must elapse for it to exit the region I. For validation purposes only, two other attributes are added to the environment:

- An attribute *trains_to_I*, a multiset of time values, where a value r represents a train which will enter I in time r. A value 0 indicates that the train has entered the intersection.
- An attribute *gpos* which models the state of the gate accurately, where *lowering*(r) (resp. *raising*(r)) means that the gate is being lowered (resp. raised) and will be down (resp. up) in time r.

We define the environment as an object in a class *env* in the same module *XING* by

sort *Gateposition*
ops *lowering*, *raising* : *Time* $\rightarrow$ *Gateposition*
ops ϵ_1, ϵ_2 : $\rightarrow$ *Time*
class *env* | *trains*, *trains_to_I* : *Multi_time*, *gpos* : *Gateposition*
msgs *lower*, *raise* : $\rightarrow$ *Msg*

Messages from the crossing object are treated as follows:

$$\textbf{eager } (lower)\langle e : env | gpos : raising(x_r)\rangle \longrightarrow \langle e : env | gpos : lowering(time_lower)\rangle$$

$$\textbf{eager } (raise)\langle e : env | gpos : lowering(x_r)\rangle \longrightarrow \langle e : env | gpos : raising(time_raise)\rangle$$

$$\textbf{eager } (raise)\langle e : env | gpos : raising(x_r)\rangle \longrightarrow \langle e : env | gpos : raising(x_r)\rangle$$

$$\textbf{eager } (lower)\langle e : env | gpos : lowering(x_r)\rangle \longrightarrow \langle e : env | gpos : lowering(x_r)\rangle.$$

The *enterR* messages are created arbitrarily, but not whenever possible (therefore, the rule is not eager):

$$\langle e : env | trains : m, trains_to_I : m'\rangle \longrightarrow \langle e : env | trains : m\ \ x_r, trains_to_I : m'\ \ y_r\rangle(enterR)$$
$$\textbf{if } \epsilon_1 \leq x_r \leq \epsilon_2 \text{ and } R_to_I \leq y_r < x_r.$$

Whenever a train leaves I, an *exitI* message must be sent:

$$\textbf{eager } \langle e : env | trains : m\ 0, trains_to_I : m'\ 0\rangle \longrightarrow \langle e : env | trains : m, trains_to_I : m'\rangle(exitI).$$

Time acts on an *env* object according to the following rules:

$$\textbf{eager } \delta(\langle e : env | trains : m', trains_to_I : m', gpos : lowering(y_r)\rangle, x_r) \longrightarrow \langle e : env | trains : m' \dot{-} x_r, trains_to_I : m' \dot{-} x_r, gpos : lowering(y_r \dot{-} x_r)\rangle$$

$$\textbf{eager } \delta(\langle e : env | trains : m', trains_to_I : m', gpos : raising(y_r)\rangle, x_r) \longrightarrow \langle e : env | trains : m' \dot{-} x_r, trains_to_I : m' \dot{-} x_r, gpos : raising(y_r \dot{-} x_r)\rangle.$$

3.4 The Combined System

Intuitively, the system can proceed in time until either the gate is up and a train could have reached the position where the gate should be lowered, or a train must exit region I, or a train enters R. Note that in this example, we use prefix notation for the symbol $\frown$.

$$tick_1 : \frown(\langle x : Xing | trains : m, gstate : up\rangle\langle e : env | trains : m'\rangle) \longrightarrow \frown(\delta(\langle x : Xing | trains : m, gstate : up\rangle\langle e : env | trains : m'\rangle, x_r))$$
$$\textbf{if } (m == \emptyset_t) \text{ or } (x_r \leq least(m) \dot{-} time_lower \text{ and } x_r \leq least(m')).$$

In case the gate is down, the system can proceed until a train leaves the region I, which is when a train value in the environment is 0:

$$tick_2 : \urcorner(\langle x : Xing | trains : m, gstate : down \rangle \langle e : env | trains : m' \rangle) \longrightarrow$$
$$\urcorner(\delta(\langle x : Xing | trains : m, gstate : down \rangle \langle e : env | trains : m' \rangle, x_r))$$
$$\textbf{if } x_r \leq least(m').$$

The fact that time increases non-deterministically by $x_r \leq least(m') \dot{-} time_lower$ and $x_r \leq least(m')$ instead of by $x_r = \min(least(m) \dot{-} time_lower, least(m'))$ allows non-deterministic rewrites where $enterR$ messages are sent at arbitrary times.

In general, time acts independently on each object:

$$\delta(yz, x_r) = \delta(y, x_r)\, \delta(z, x_r)$$

for $y, z : Configuration$. Note that, given the eagerness with which all messages are processed and the δ-rules are applied, there are never any messages present in a configuration when a tick rule is applied.

Once we have a specification of this kind, we can execute it, to simulate the behavior of the intended system, and to uncover some possible bugs in the specification itself. This form of symbolic simulation can already prove certain properties of the system as sequents derivable from the specification. In addition, the initial model of the rewriting logic specification [13] provides a precise mathematical model against which formal statements about the behavior that the system must exhibit can be verified. For example, one could show that whenever a train is in the intersection (represented by a value 0 in the environments $trains_to_I$ attribute), the gates are down. One way of proving this is to show that whenever the initial configuration rewrites to a state of the form

$$\urcorner(\langle x : Xing \rangle \langle e : env | trains_to_I : 0 \;\; m, gpos : g \rangle M)$$

(for M a multiset of messages), then the value of g is $lowering(0)$.

4 Rewriting Logic as a Semantic Framework for Real-Time Systems

This section illustrates how a variety of models of real-time systems have a natural translation into rewriting logic. Apart from Section 4.1, we concentrate on rewriting logic specifications that can be used directly for "simulating" the corresponding systems. The method indicated in Section 4.1 can be used to reason about quantitative properties of the systems so specified.

4.1 Timed Automata

Omitting details about initial states and acceptance conditions, a timed automaton (see, e.g., [3]) consists of:

- a finite alphabet Σ,
- a finite set S of states,

- a finite set C of clocks,
- a set $\Phi(C)$ of clock constraints defined inductively by
$$\phi := c \leq k \mid k \leq c \mid \neg\phi \mid \phi_1 \wedge \phi_2$$
where c is a clock in C and k is a constant in the set of nonnegative rationals, and
- a set $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ of transitions. The tuple $\langle s, s', a, \lambda, \phi \rangle$ represents a transition from state s to state s' on input symbol a. The set $\lambda \subseteq C$ gives the clocks to be reset with this transition, and ϕ is a clock constraint over C.

Given a timed word (i. e. a sequence of tuples $\langle a_i, r_i \rangle$ where a_i is an input symbol and r_i is the time at which it occurs), the automaton starts at time 0 with all clocks initialized to 0. As time advances the values of all clocks change, reflecting the elapsed time; that is, the state of the automaton can change not only by the above transitions, but also by the passage of time, with all the clocks being increased by the same amount. At time r_i the automaton changes state from s to s' using some transition of the form $\langle s, s', a_i, \lambda, \phi \rangle$ reading input a_i, if the current values of the clocks satisfy ϕ. With this transition the clocks in λ are reset to 0, and thus start counting time again.

A timed automaton can be naturally represented in rewriting logic. The time domain and its associated constraints $\Phi(C)$ are equationally axiomatized in an abstract data type satisfying the theory *TIME*. Then, the tuple $\langle s, c_1, \ldots, c_n \rangle$ represents an automaton in state s such that the values of the clocks in C are $c_1, \ldots, c_n$. Each transition $\langle s, s', a, \lambda, \phi \rangle$ is then expressed as a rewrite rule
$$a : \langle s, c_1, \ldots, c_n \rangle \longrightarrow \langle s', c'_1, \ldots, c'_n \rangle \text{ if } \phi(c_1, \ldots, c_n)$$
where $c'_i = 0$ if $c_i \in \lambda$, and $c'_i = c_i$ otherwise. In addition, a rule
$$tick : \langle x, c_1, \ldots, c_n \rangle \longrightarrow \langle x, c_1 + x_r, \ldots, c_n + x_r \rangle$$
(where $x, x_r, c_1, \ldots, c_n$ all are variables) is added to represent the elapse of time. It is easy to show that the resulting rewriting logic specification faithfully represents the timed automaton.

Using the ideas of "time as an action" and of tick rules advancing time for a system as a whole, we can give a somewhat more detailed specification of a timed automaton. We leave the transition rewrite rules unchanged, but replace the rewrite rule
$$tick : \langle x, c_1, \ldots, c_n \rangle \longrightarrow \langle x, c_1 + x_r, \ldots, c_n + x_r \rangle$$
by the following two rules:
$$tick : \langle s, \widehat{c_1, \ldots, c_n} \rangle \longrightarrow \delta(\langle x, \widehat{c_1, \ldots, c_n} \rangle, x_r),$$
$$\delta : \delta(\langle x, c_1, \ldots, c_n \rangle, x_r) \longrightarrow \langle x, c_1 + x_r, \ldots, c_n + x_r \rangle.$$
That is, we decompose the original tick into two steps, namely the action of time and the subsequent passage to the resulting new state. In this way, we have two possibilities available to us. On the one hand, time elapse properties of the timed automaton such as a rewrite $\delta(\langle s_0, \ldots \rangle, 50) \longrightarrow \langle s_n, \ldots \rangle$ can be

proved about the system. On the other hand, for simulation purposes the rewrite rules can be used to derive all the possible runs of the automaton in the form:

$$\langle s_0, \widehat{0, \ldots, 0} \rangle \xrightarrow{tick} \delta(\langle s_0, \widehat{0, \ldots, 0} \rangle, r) \xrightarrow{\delta} \langle s_0, \widehat{r, \ldots, r} \rangle \xrightarrow{a} \langle s_1, \widehat{r, \ldots, 0} \rangle \xrightarrow{tick} \cdots$$

4.2 Hybrid Automata

The time model of *hybrid automata* [2] is the real numbers. However, to get a computable data type, we should replace the reals by a computable subfield $\mathbb{R}$, such as the algebraic real numbers. A hybrid automaton is given by a tuple $\langle V_D, Loc, Act, Inv, Edg \rangle$ (we omit acceptance conditions and initial states) where:

- V_D is a finite set of real-valued data variables, defining the data space Σ_D, that is, Σ_D is the function space $[V_D \to \mathbb{R}]$.
- Loc is a finite set of *locations* (corresponding to "states" in untimed automata).
- Act is a labeling function that assigns to each location $l \in Loc$ a set Act_l of *activities*. An activity is a function from $\mathbb{R}_+$ to Σ_D. A system in state $\langle l, \overline{v} \rangle$ evolves to $\langle l, f(r) \rangle$ in time r whenever f is an activity of location l such that $\overline{v} = f(0)$.
- Inv is a labeling function that assigns to each location $l \in Loc$ an *invariant* $Inv(l) \subseteq \Sigma_D$.
- Edg is a finite set of *transitions*. Each transition $e = (l, \mu, l')$ consists of a source location l, a target location l', and a transition relation $\mu \subseteq \Sigma_D^2$. For each location l there is a *stutter transition* (l, Id, l) where $Id = \{(\overline{v}, \overline{v}) \mid \overline{v} \in \Sigma_D\}$.

At any time instant, the state $\langle l, \overline{v} \rangle$ of a hybrid system specifies a control location and values for all data variables, i.e., the state space is $Loc \times \Sigma_D$. The state can change in two ways: (1) by an instantaneous transition that changes the entire state according to the transition relation, or (2) by elapse of time that changes only the values of data variables in a continuous manner according to the activities of the current location. The system may stay at a location only if the invariant at the location is true. The invariants of a hybrid automaton thus enforce the progress of the underlying discrete transition system: some transition must be taken before the invariant of the location is false.

As in [13], where a transition τ from state s to state s' is modeled by a rewrite rule $\tau : s \longrightarrow s'$, it is assumed that the transitions Edg can be expressed by rewrite rules

$$\langle l, \overline{v} \rangle \longrightarrow \langle l', \overline{v'} \rangle.$$

To specify the continuous behavior of a system, one needs to know the maximum time such that, given a state $\langle l, \overline{v} \rangle$, control can stay at location l performing the activity f without violating the invariant of location l. We

assume that, for each location l, this is given by a function

$$max_stay_l : Act_l \to Time_\infty$$

The tick rules of the system are given by

$$tick : \langle l, f(0)\rangle \longrightarrow \langle l, f(x_r)\rangle \textbf{ if } x_r \leq max_stay_l(f)$$

for all locations l and for each activity f in l.

4.3 Timed Transition Systems

A *timed transition system* (TTS) [11], whose time domain is the set $\mathbb{N}$ of natural numbers, is a transition system (with a finite number of transitions) where each transition τ is equipped with a "lower bound" $l_\tau \in \mathbb{N}$ and an "upper bound" $u_\tau \in \mathbb{N} \cup \{\infty\}$. A transition τ cannot be taken if it hasn't been enabled uninterruptedly for at least l_τ time units, and if τ is enabled at time n, then either τ is disabled or taken somewhere in the interval $[n, n+u_\tau]$. In the TTS model, there is no continuous change of state.

As for hybrid automata, we assume that the underlying untimed transition system can be specified in rewriting logic as shown in [13]. A TTS is represented in rewriting logic by just adding to each state one clock for each transition, where the clock c_i has value *nil* if transition τ_i is not enabled, and t_i if the transition has been enabled continuously for time t_i (without being taken). The symbol *nil* is therefore an element of a supersort of the natural numbers, satisfying the equation $nil + x = nil$ for $x : Nat$. We also assume that for each transition τ_i, there is a predicate $enabled_i$ such that $enabled_i(s)$ is true if transition τ_i is enabled on state s and false otherwise.

A state of the system is thus represented as a term $\langle s, c_1, \ldots, c_n\rangle$, where s is the state of the transition system, and the c_i's are the clocks. A transition

$$\tau_i : s \longrightarrow s'$$

in the TTS is modeled by a rewrite rule

$$\tau_i : \langle s, c_1, \ldots, c_n\rangle \longrightarrow \langle s', c_1', \ldots, c_n'\rangle \textbf{ if } c_i \geq l_{\tau_i}$$

where for all $j = 1, \ldots, n$,

$c_j' =$ **if** $not(enabled_j(s'))$ **then** *nil* **else if** $c_j == nil$ or $i == j$ **then** 0 **else** c_j.

Time can elapse if no transition must be taken:

$$tick : \langle x, c_1, \ldots, c_n\rangle \longrightarrow \langle x, c_1 + x_r, \ldots, c_n + x_r\rangle \textbf{ if } \bigwedge_i (c_i + x_r \leq u_{\tau_i} \text{ or } c_i == nil)$$

4.4 Phase Transition Systems

Phase transition systems (PTS's) [11] extend timed transition systems to hybrid systems. In a PTS the set V of variables defining the state space is divided into two parts: the set V_c of continuous real variables, again, for $\mathbb{R}$ a computable subfield of the reals, and the set V_d of discontinuous real variables. The continuous variables change their values with the elapse of time according to activities of the form

$$P \Longrightarrow \dot{v}_{c_i} t,$$

where P is a Boolean expression over the discontinuous variables and t is a term over V. We assume without loss of generality that the continuous behavior of each continuous variable is described by one such activity.

The set $\mathcal{T}$ of instantaneous transitions is, as in the TTS case, equipped with upper and lower bounds.

To give a rewriting logic specification, the set V of variables must be finite, and the following functions must be computable:

- the effect of letting time act on a continuous variable v_{c_i}, given the current state, is assumed given by a function
$$g_i : \mathbb{R}^{m+n}, \mathbb{R}_+ \to \mathbb{R}$$
where $g_i(v_{c_1}, \ldots, v_{c_m}, v_{d_1}, \ldots, v_{d_n}, r)$ gives the value of v_{c_i} when the system proceeds r time units from the state $\langle v_{c_1}, \ldots, v_{c_m}, v_{d_1}, \ldots, v_{d_n} \rangle$, and
- a function
$$f : \mathbb{R}^{m+n} \to \mathbb{R}$$
which takes a state as argument and gives the maximum time the system can proceed without changing the enabledness of a transition.

We represent the state of a PTS as the term

$$\langle v_{c_1}, \ldots, v_{c_m}, v_{d_1}, \ldots, v_{d_n}, c_1, \ldots, c_k \rangle$$

where v_{c_i} (v_{d_i}) denotes the current value of the PTS variable v_{c_i} (resp. v_{d_i}) and c_i indicates (as in the timed transition system case) for how long transition τ_i has been enabled without being taken.

The discrete parts of the system are given as in the TTS case, i.e.,

$$\tau_i : \langle \overline{v_c}, \overline{v_d}, c_1, \ldots, c_k \rangle \longrightarrow \langle \overline{v'_c}, \overline{v'_d}, c'_1, \ldots, c'_k \rangle \textbf{ if } l_{\tau_i} \leq c_i$$

where for all $j = 1, \ldots, n$,

$c'_j =$ **if** $not(enabled_j(s'))$ **then** nil **else if** $c_j == nil$ or $i == j$ **then** 0 **else** c_j.

Specifying the elapse of time we must ensure that:

- all continuous variables are updated,
- if any enabling condition changes, then the corresponding clocks are updated, and
- time cannot elapse beyond the point at which an enabled transition must be taken.

The following tick rules handle these cases:

$$tick_1 : \langle v_{c_1}, \ldots, v_{c_m}, \overline{v_d}, c_1, \ldots, c_k \rangle \longrightarrow$$
$$\langle g_1(\overline{v_c}, \overline{v_d}, x_r), \ldots, g_m(\overline{v_c}, \overline{v_d}, x_r), \overline{v_d}, c_1 + x_r, \ldots, c_k + x_r \rangle$$
$$\textbf{if } \textstyle\bigwedge_i (c_i + x_r \leq u_{\tau_i} \text{ or } c_i == nil) \text{ and } x_r < f(\overline{v_c}, \overline{v_d})$$

$$tick_2 : \langle v_{c_1}, \ldots, v_{c_m}, \overline{v_d}, c_1, \ldots, c_k \rangle \longrightarrow$$
$$\langle g_1(\overline{v_c}, \overline{v_d}, x_r), \ldots, g_m(\overline{v_c}, \overline{v_d}, x_r), \overline{v_d}, c'_1, \ldots, c'_k \rangle$$
$$\textbf{if } \textstyle\bigwedge_i (c_i + x_r \leq u_{\tau_i} \text{ or } c_i == nil) \text{ and } x_r = f(\overline{v_c}, \overline{v_d})$$

where for all $j = 1, \ldots, n$,

$$c'_j = \textbf{ if } not(enabled_j(s')) \textbf{ then } nil \textbf{ else if } c_j == nil \textbf{ then } 0 \textbf{ else } c_j + x_r.$$

In the rule $tick_2$, time is advanced until an enabling condition changes; therefore, all enabling conditions must be reevaluated to their new values $c'_1, \ldots, c'_k$.

4.5 Timed Petri Nets

Petri nets have been extended to model real-time systems in different ways (see e. g. [1,17]). Three of the most used time extensions are the following:

(i) Each *transition* α has an associated time interval $[l_\alpha, r_\alpha]$. A transition fires as soon as it can, but the resulting tokens are delayed, i. e., they are not visible in the system until some time $t \in [l_\alpha, u_\alpha]$ after the transition fires.

(ii) Each *place* p has a duration t_p. A token at place p cannot participate in a transition until it has been at p for at least time t_p.

(iii) Each transition α is associated with a time interval $[l_\alpha, u_\alpha]$ and the transition α cannot fire before it has been continuously enabled for at least time l_α. Also, the transition α cannot have been enabled continuously for more than time u_α without being taken.

We only treat the first two cases. The third case can be given a treatment similar to that of timed transition systems.

The translation into rewriting logic of these first two cases is based on the rewriting logic representation of untimed Petri nets given in [13], where the state of a Petri net is represented by a multiset of places—where if place p has multiplicity n we interpret this as the presence of n tokens at the place—and where the transitions correspond to rewrite rules on the corresponding multisets of pre- and post-places.

In our representation, a token at a place p is denoted by a term

$$[p].$$

A token that will be available at place p in time r is represented by the term

$$dly(p, r).$$

A state of a timed Petri net is a multiset of these two forms of placed tokens, where multiset union is represented by juxtaposition.

Time acts on a placed token as follows:

$$\delta([p], x_r) \longrightarrow [p]$$

$$\delta(dly(p, x_r), x_r) \longrightarrow [p]$$

and time distributes over multisets of placed tokens:

$$\delta(\emptyset, x_r) = \emptyset$$

$$\delta(xy, x_r) = \delta(x, x_r)\ \delta(y, x_r),$$

where x, y range over multisets of placed tokens and $\emptyset$ denotes the empty multiset.

Transitions are represented by rewrite rules. In the first case of timed Petri nets, also known as *interval timed Petri nets* [1], each transition α has an associated interval $[l_\alpha, u_\alpha]$. Assume that the transition α consumes two tokens from place a, and one token from place b, and produces one token each at places c and d. Since the duration of the transition is any time in the interval $[l_\alpha, u_\alpha]$, the resulting tokens are not visible for a time within this interval. Hence the transition α can be represented by the following rewrite rule:

$$\textbf{eager } [a][a][b] \longrightarrow dly(c, x_r)\ dly(d, x_r) \textbf{ if } l_\alpha \leq x_r \leq u_\alpha.$$

In the second version of timed Petri nets, each place p has an associated duration t_p, and a token must have been at a place p for at least time t_p before it can be used in any transition. This is equivalent to saying that the produced token cannot be visible before time t_p after the producing transition took place. Hence the transition that consumes two tokens from place a and one from place b, and which produces one token each at c and d is represented in rewriting logic by the rule

$$\textbf{eager } [a][a][b] \longrightarrow dly(c, t_c)\ dly(d, t_d).$$

As usual, the elapse of time (in both versions) is modeled by tick rules. In order to ensure that time does not proceed beyond the time when a transition could fire (that is, when time has acted on a token $dly(p, r)$ for time r), the operator *least_tick* is used. It takes as argument a multiset of placed tokens, returns the time until one or more non-available tokens become available, and is defined in the following way:

$$\begin{aligned} least_tick(\emptyset) &= \infty \\ least_tick([p]) &= \infty \\ least_tick(\delta([p], x_r)) &= 0 \\ least_tick(\delta(dly(p, x_r), y_r)) &= x_r \dot{-} y_r \\ least_tick(x\ y) &= \min(least_tick(x), least_tick(y)). \end{aligned}$$

The tick rule then allows time to elapse until the first *dly*-token becomes visible[3]:

$$tick : \widehat{N} \longrightarrow \delta(N, \widehat{least_tick}(N)).$$

In this case the operator $\widehat{\ }$ is needed. Otherwise, time could elapse only in parts of the whole system N.

In both versions of timed Petri nets transitions are supposed to fire as soon as possible. This is accomplished by the strategy described in Section 2.4 that triggers all eager instantaneous rules until none of these can be applied, followed by one application of the tick rule. No explicit eagerness is required for the rules in R_δ, since time will not elapse (by a time value greater than 0) if these are not applied whenever enabled.

[3] A more general tick rule $\widehat{N} \longrightarrow \widehat{\delta(N, r)}$ **if** $r \leq least_tick(N)$ would not do any good, since nothing can be done until the next unavailable token becomes available.

5 Relationship to Timed Rewriting Logic

In this section we investigate the relationship between timed rewriting logic and the described framework for specifying real-time systems directly in rewriting logic. After briefly introducing TRL in Section 5.1, we propose in Section 5.2 a translation from TRL into rewriting logic. In this translation, the translation of any derivable TRL-sequent in a TRL theory is derivable in the corresponding rewriting logic theory. The converse is in general not true. We explain the reasons for this discrepancy in Section 5.3. They are due to some conceptual differences between TRL and our method of specifying real-time systems in rewriting logic. However, in Section 5.5 we show how the proposed translation in many cases can be extended to simulate systems specified in TRL by means of their rewriting logic translations.

5.1 Timed Rewriting Logic

Rewriting logic has been extended by Kosiuczenko and Wirsing to handle real-time systems in their *timed rewriting logic* (TRL) [9,10]. TRL has been shown well-suited for giving object-oriented specifications of complex hybrid systems such as the steam-boiler [18] and has been illustrated by a number of specifications of simpler real-time systems. A translation into ordinary rewriting logic can illuminate the conceptual relationships between both formalisms. Also, since rewriting logic seems easier to implement than TRL and in fact several such implementations already exist, such a translation can also provide a convenient path to make TRL specifications executable.

In TRL each rewrite step is labeled with a time stamp. TRL rules are sequents of the form $t \xrightarrow{r} t'$, for r a ground term of sort *Time*. Their intuitive meaning is that t evolves to t' in time r.

More specifically, assume given a TRL theory, i.e., a set of equations and timed rewrite rules satisfying the theory *TIME*[4]. Then, the set of derivable sequents consists of all rules in the specification, and all sequents which can be derived by equational reasoning and by using the deduction rules in Figure 1, where $\mathcal{V}(t)$ denotes the set of free variables in t.

This deduction system extends and modifies the rules of deduction in rewriting logic with time stamps as follows:

- Reflexivity is dropped as a general axiom, to allow specifying hard real-time systems. Reflexivity would not allow describing hard real-time systems since (parts of) the system could stay idle for an arbitrary long period of time. For specifying soft real-time systems particular reflexivity axioms can be added.
- Transitivity yields the addition of the time stamps. If t_1 evolves to t_2 in time r_1 and t_2 evolves to t_3 in time r_2, then t_1 evolves to t_3 in time $r_1 + r_2$.
- The synchronous replacement rule enforces uniform time elapse in all com-

[4] They impose in some cases further requirements, such as *TIME* being an Archimedean monoid. This could of course be easily accomodated.

Timed transitivity (TT):

$$\frac{t_1 \xrightarrow{r_1} t_2 \quad t_2 \xrightarrow{r_2} t_3}{t_1 \xrightarrow{r_1+r_2} t_3}$$

Synchronous replacement (SR):

$$\frac{t_0 \xrightarrow{r} t_0',\ t_{i_1} \xrightarrow{r} t_{i_1}', \ldots, t_{i_k} \xrightarrow{r} t_{i_k}'}{t_0(t_1/x_1, \ldots, t_n/x_n) \xrightarrow{r} t_0'(t_1'/x_1, \ldots, t_n'/x_n)}$$

where $\{x_{i_1}, \ldots, x_{i_k}\} = \mathcal{V}(t_0) \cap \mathcal{V}(t_0')$.

Compatibility with equality (EQ):

$$\frac{t_1 = u_1,\ r_1 = r_2,\ t_2 = u_2,\ t_1 \xrightarrow{r_1} t_2}{u_1 \xrightarrow{r_2} u_2}$$

Renaming of variables (RV):

$$x \xrightarrow{r} x \qquad \text{for all } x \in X, r \in T_{\Sigma_{Time}}$$

Fig. 1. *Deduction rules in timed rewriting logic.*

ponents of a system: a system rewrites in time r iff all its components do so. Synchronous replacement combined with irreflexivity also induces maximal parallelism, which means that no component of a process can stay idle.

- The renaming rule assures that timed rewriting is independent of the names of variables. Observe that the renaming axiom does not imply that $t \xrightarrow{r} t$ holds for all terms t.

Example 5.1 From the timed rewrite specification $\{f(x) \xrightarrow{1} g(x), g(x) \xrightarrow{1} h(x), a \xrightarrow{2} b\}$, where time is modeled by the natural numbers, the sequent $f(a) \xrightarrow{2} h(b)$ can be deduced by first deducing $f(x) \xrightarrow{2} h(x)$ using transitivity and then applying the synchronous replacement rule. Due to the lack of $\xrightarrow{r}$-reflexivity, neither $a \xrightarrow{2} a$ nor $f(a) \xrightarrow{2} h(a)$ are derivable. Note that $f(a) \xrightarrow{2} h(b)$ can not be deduced without using the synchronous replacement rule.

5.2 Timed Rewriting Logic in Rewriting Logic

In this section we define a mapping from timed rewriting logic to rewriting logic. In other words, given a TRL specification $\mathcal{T}$, there is a mapping $\mathcal{M}$ sending $\mathcal{T}$ to $\mathcal{M}(\mathcal{T})$, and sending TRL sequents $t \xrightarrow{r} t'$ to sequents $\mathcal{M}(t \xrightarrow{r} t')$ in rewriting logic, such that $\mathcal{T} \vdash t \xrightarrow{r} t'$ implies that $\mathcal{M}(\mathcal{T}) \vdash \mathcal{M}(t \xrightarrow{r} t')$ for all terms t, t'.

We restrict our treatment to TRL theories where no extra variables are introduced in the righthand side of a rule. The reason for this restriction is that if $f(x) \xrightarrow{2} g(x, y)$ and $g(x, y) \xrightarrow{2} h(y)$ are two rules, any system t' that appears in $h(t)$ as a result of the second rule, must have evolved for time 2 from a system t in $g(u, t)$. However, by transitivity of the rules, the sequent $f(x) \xrightarrow{4} h(y)$ is derivable, which means that *any* system t could replace y in $h(y)$, including the systems which have not evolved for time 2.

The idea of the translation from TRL to rewriting logic is that a TRL sequent $t \xrightarrow{r} t'$ ("t evolves in time r to t'") maps to a rewriting logic sequent $\delta(t,r) \longrightarrow t'$ ("if time has acted on t for time r, then it rewrites to t'") for ground terms t, t'. If t contains a variable x, the subsystem t_i by which x is instantiated in t should have evolved r time units in t'. A sequent $t \xrightarrow{r} t'$ is therefore mapped to

$$\delta(t,r) \longrightarrow t'(\delta(x_1,r)/x_1, \ldots, \delta(x_n,r)/x_n)$$

for the set $\{x_1, \ldots, x_n\} \supseteq \mathcal{V}(t') \subseteq \mathcal{V}(t)$ of variables.

The mapping $\mathcal{M}$ from TRL-sequents to rewriting logic sequents is then given by

$$\mathcal{M}(t \xrightarrow{r} t') = \delta(t,r) \longrightarrow t'(\delta(x_1,r)/x_1, \ldots, \delta(x_n,r)/x_n)$$

where the set of free variables in t' is a subset of $\{x_1, \ldots, x_n\}$.

This map $\mathcal{M}$ is extended to a map sending a TRL theory $\mathcal{T} = \langle \Sigma, E, R \rangle$ satisfying the above restrictions of no extra variables in righthand sides to a rewriting logic theory $\mathcal{M}(\mathcal{T})$ by

$$\mathcal{M}(\Sigma) = \Sigma \cup \{\delta : s, \mathit{Time} \to s \mid s \in \mathit{sorts}(\Sigma)\}$$

$$\mathcal{M}(E) = E \cup \{(\forall x\!:\!s, x_r, y_r\!:\!\mathit{Time})\ \ \delta(x,0) = x,\ \delta(\delta(x,x_r),y_r) = \delta(x, x_r + y_r) \mid s \in \mathit{sorts}(\Sigma)\}$$

$$\mathcal{M}(R) = \{\mathcal{M}(\rho) \mid \rho \in R\}$$

Therefore, $\mathcal{M}$ can naturally be understood as a map of logics. Specifically, as a map $\mathcal{M} : \mathit{TRL} \longrightarrow \mathit{RWL}$ from the entailment system [15] of TRL to that of rewriting logic.

Theorem 5.2 *Let $\mathcal{T}$ be a TRL specification and let $\mathcal{M}$ be defined as above. Then for all terms t, t', r*

$$\mathcal{T} \vdash t \xrightarrow{r} t' \text{ implies } \mathcal{M}(\mathcal{T}) \vdash \mathcal{M}(t \xrightarrow{r} t').$$

As a corollary of this theorem, which can be easily proved by induction on the size of the proof $t \xrightarrow{r} t'$, we obtain that $\mathcal{T} \vdash t \xrightarrow{r} t'$ implies $\mathcal{M}(\mathcal{T}) \vdash \delta(t,r) \longrightarrow t'$ for all ground terms t, t', and r.

Example 5.3 (Example 5.1 cont.) The translation of the TRL specification in Example 5.1 is given by the equations for the action δ and the rules

$$\{\delta(f(x),1) \longrightarrow g(\delta(x,1)), \delta(g(x),1) \longrightarrow h(\delta(x,1)), \delta(a,2) \longrightarrow b\}.$$

The sequent $\delta(f(a),2) \longrightarrow h(b)$ corresponding to $f(a) \xrightarrow{2} h(b)$ is obtained by $\delta(f(a),2) \longrightarrow \delta(g(\delta(a,1)),1) \longrightarrow h(\delta(a,2)) \longrightarrow h(b)$, where use of the equation $\delta(\delta(x,x_r),y_r) = \delta(x, x_r + y_r)$ is not shown explicitly.

5.3 Differences Between TRL and its Rewriting Logic Translation

Even though $t \xrightarrow{r} t'$ implies $\delta(t,r) \longrightarrow t'$ for ground terms, the converse is not necessarily true. In this section the differences between deduction in TRL and in its translation into rewriting logic are outlined.

5.3.1 *Zero-Time Idling*

In the rewriting logic translation, a TRL sequent $t \xrightarrow{0} t$ translates to $\delta(t,0) \longrightarrow t(\delta(x_1,0)/x_1,\ldots,\delta(x_n,0)/x_n)$, which due to the axiom $\delta(x,0) = x$ is equal to $t \longrightarrow t$, which is always deducible in rewriting logic. However, in TRL, $t \xrightarrow{0} t$ is not necessarily valid. This obviously indicates a difference between both systems, since the notion of "zero-time idling" is always available in our approach but not in TRL.

5.3.2 *Non-Right-Linear Rules*

Given the TRL theory $\{f(x) \xrightarrow{2} g(x,x), a \xrightarrow{2} b, a \xrightarrow{2} c\}$, the term $f(a)$ rewrites to either $g(b,b)$ or $g(c,c)$ in time two, but will *not* rewrite to $g(b,c)$. In the rewriting logic translation $\{\delta(f(x),2) \longrightarrow g(\delta(x,2),\delta(x,2)), \delta(a,2) \longrightarrow b, \delta(a,2) \longrightarrow c\}$ there is a rewrite $\delta(f(a),2) \longrightarrow g(\delta(a,2),\delta(a,2)) \longrightarrow g(b,c)$.

The difference depends on how one models the fork of a process. The rule $f(x) \xrightarrow{r} g(x,x)$ can be understood as a fork of the (sub)process t in the system $f(t)$. In the TRL setting, the actual "fork" (the point in time when the two instances of the process x can behave independently of each other) is taking place at the end of the time period of length r in the rule. In the rewriting logic setting, the "forking" took place at the beginning of the time period of duration r [5].

5.3.3 *Problems Related to Synchronicity in TRL*

Another aspect in which TRL and our rewriting logic translation are different is illustrated by the following TRL specification:

$$\{f(a,y) \xrightarrow{2} g(a,y), g(x,y) \xrightarrow{2} h(x,y), h(x,c) \xrightarrow{2} k(x,c), a \xrightarrow{4} d, b \xrightarrow{4} c\}.$$

Due to the strong synchronicity requirements in TRL, $f(a,b)$ cannot be rewritten, even though the b (in the place of y), and a (for x), could be rewritten in time 4. In many cases, it would however be natural to assume that the system represented by $f(a,b)$ rewrites to $k(d,c)$ in time 6.

In the rewriting logic translation, $\delta(f(a,b),6)$ rewrites to $k(d,c)$.

5.4 *Aging in TRL*

To overcome the strong requirements of synchronicity in TRL, which caused the differences in Sections 5.3.2 and 5.3.3, the special symbol *age* is introduced in [9,10]. It aims at making a term t, which rewrites in time r', "accessible" to synchronous rewrites in time r with $r' \geq r$, by making it visible as $age(t,r)$.

Formally, with aging, the following two deduction rules are added to the TRL deduction rules given in Figure 1. In both deduction rules, $t \xrightarrow{r+r'} t'$ is

[5] Note that in the rewriting logic setting, adding a rule $\delta(k(x),2) \longrightarrow f(\delta(x,2))$ to the system above gives $\delta(k(x),4) \longrightarrow g(\delta(x,4),\delta(x,4))$, hence a "fork" which took place too early. Such behavior can be avoided by requiring that the variable x in the rule $\{\delta(f(x),2) \longrightarrow g(\delta(x,2),\delta(x,2))$ has a "non-δ-sort" (see Section 5.5).

assumed to be a timed rewrite *rule* in the specification.

$$age_1 : \frac{}{t \xrightarrow{r} age(t,r)} \qquad age_2 : \frac{}{age(t,r) \xrightarrow{r'} t'}$$

The *age* operator also satisfies the axiom $age(age(t,r),r') = age(t,r+r')$ for all terms t and time values r, r'.

With aging, the "fork" differences disappear, since (assuming $g(x,y) \xrightarrow{0} g(x,y)$) we have $f(a) \xrightarrow{2} g(age(a,2), age(a,2)) \xrightarrow{0} g(b,c)$ for the system in the example of Section 5.3.2, and the strong synchronicity is loosened, as illustrated by the fact that in Section 5.3.3, $f(a,b) \xrightarrow{6} k(d,c)$ is derivable, since $f(a,b) \xrightarrow{2} g(a, age(b,2)), g(a, age(b,2)) \xrightarrow{2} h(age(a,2),c)$, and $h(age(a,2),c) \xrightarrow{2} k(d,c)$ are derivable.

Unfortunately, the deduction rules for aging lead to counterintuitive results, as illustrated by the following example:

Example 5.4 Given the TRL theory $\{f(x) \xrightarrow{2} g(x), f(b) \xrightarrow{2} g(c), a \xrightarrow{2} b\}$, one would expect that $f(a) \xrightarrow{2} g(c)$ is *not* derivable. However, $f(x) \xrightarrow{2} age(f(x),2)$ and $age(f(x),2) \xrightarrow{0} g(x)$ are derivable, and so are $f(b) \xrightarrow{2} age(f(b),2)$ and $age(f(b),2) \xrightarrow{0} g(c)$.

The sequents $f(x) \xrightarrow{2} age(f(x),2)$ and $a \xrightarrow{2} b$ give $f(a) \xrightarrow{2} age(f(b),2)$ by synchronous replacement, which in turn rewrites to $g(c)$ using $age(f(b),2) \xrightarrow{0} g(c)$. Transitivity gives the undesired sequent $f(a) \xrightarrow{2} g(c)$.

We can summarize the situation as follows. We have seen that the rewriting translation of a TRL theory $\mathcal{T}$ is looser than $\mathcal{T}$ itself, in some cases with some pleasant consequences. If we attempt to tighten the correspondence between both systems by adding aging rules to TRL, we get indeed closer, but we unfortunately encounter paradoxical examples in the reformulation of TRL.

5.5 Simulation of TRL Theories in Rewriting Logic

In spite of the above mentioned discrepancies between the two formal systems, in practice there are interesting examples for which a TRL theory $\mathcal{T}$ and its translation $\mathcal{M}(\mathcal{T})$ behave in exactly similar ways for ground terms, in the sense that given $t, t' \in T_\Sigma$, $r \in T_{\Sigma_{Time}}$ we have

$$\mathcal{T} \vdash t \xrightarrow{r} t' \iff \mathcal{M}(\mathcal{T}) \vdash \mathcal{M}(t \xrightarrow{r} t').$$

If we find ourselves in such a good situation, it becomes interesting to use rewriting logic not only to mirror the deduction of the TRL theory $\mathcal{T}$, but also for simulation purposes, so as to observe the behavior of the real-time system specified by $\mathcal{T}$. This can be done using ideas similar to those in Section 2.3. That is, we can extend $\mathcal{M}(\mathcal{T})$ by adding an appropriate global sort and tick rules to a rewrite theory, denoted $\mathcal{T}^t$, that will allow us to simulate the system specified by $\mathcal{T}$. However, instead of having tick rules of the form

$$\widehat{x} \longrightarrow \widehat{\delta(x, t_r)},$$

as in Section 2.3, the tick rules will now be of the form

$$\widehat{x} \longrightarrow \widehat{y} \text{ if } \delta(x, x_r) \longrightarrow y$$

and terms will be given sorts in a way such that only terms not containing δ can be substituted for y. In this way, exactly all possible TRL-rewrites can be deduced, while we in a simulation trace never will see a term containing the symbol δ (they will be hidden in the conditions allowing each rewrite).

Formally, given an order-sorted TRL theory $\mathcal{T} = \langle\langle S, \leq, \Sigma\rangle, E, R\rangle$, the translation $\mathcal{T}^t$ is defined as follows:

$$\begin{aligned}
S^t &= S \cup \{s^t \mid s \in S\} \cup \{\mathit{whole_system}\} \\
\leq^t &= \text{the reflexive-transitive closure of } \leq \cup \{s \leq s^t \mid s \in S\} \\
\Sigma^t &= \Sigma \cup \{f : s_1^t, \ldots, s_n^t \to s^t \mid f : s_1, \ldots, s_n \to s \in \Sigma \wedge n \geq 1\} \\
&\quad \cup \{\delta : s^t, \mathit{Time} \to s^t \mid s \in S\} \\
&\quad \cup \{\widehat{\ } : s^t \to \mathit{whole_system} \mid s \in S\}.
\end{aligned}$$

Since the sorts in S are only used to know which terms do not contain δ symbols, the variables in the equations and rules should be of sorts s^t ($s \in S$), hence

$$\begin{aligned}
E^t = \{&(\forall x_1 : s_1^t, \ldots, x_n : s_n^t)\ \ t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)\ \mid \\
&(\forall x_1 : s_1, \ldots, x_n : s_n)\ \ t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n) \in E\} \\
\cup \{&(\forall x_r, y_r : \mathit{Time}, x : s^t)\ \ \delta(\delta(x, x_r), y_r) = \delta(x, x_r + y_r),\ \delta(x, 0) = x \mid s \in S\}.
\end{aligned}$$

The set R^t of rules contains the translation of the TRL rules (but relaxing each sort s to s^t),

$$\begin{aligned}
\{&(\forall x_1 : s_1^t, \ldots, x_n : s_n^t)\ \ \delta(t(x_1, \ldots, x_n), r) \longrightarrow t'(\delta(x_1, r)/x_1, \ldots, \delta(x_n, r)/x_n)\ \mid \\
&(\forall x_1 : s_1, \ldots, x_n : s_n)\ \ t(x_1, \ldots, x_n) \xrightarrow{r} t'(x_1, \ldots, x_n) \in R\}
\end{aligned}$$

plus the tick rules

$$\{(\forall x : s, y : s', x_r : \mathit{Time})\ \ \widehat{x} \longrightarrow \widehat{y}\ \textbf{if}\ \delta(x, x_r) \longrightarrow y \mid s, s' \in S\}.$$

It is easy to show that

$$\mathcal{M}(\mathcal{T}) \vdash \delta(t, r) \longrightarrow t'(\delta(x_1, r)/x_1, \ldots, \delta(x_n, r)/x_n)$$

iff

$$\mathcal{T}^t \vdash \delta(t, r) \longrightarrow t'(\delta(x_1, r)/x_1, \ldots, \delta(x_n, r)/x_n)$$

since $\mathcal{T}^t$ and $\mathcal{M}(\mathcal{T})$ only differ in the sorts of terms containing δ, the sort s^t in $\mathcal{T}^t$ contains the same set of terms as the sort s in $\mathcal{M}(\mathcal{T})$, and all variables of sort s in $\mathcal{M}(\mathcal{T})$ have sort s^t in the rules and equations of $\mathcal{T}^t$. Therefore,

$$\forall r \in T_{\Sigma_{\mathit{Time}}}, \forall t, t' \in T_\Sigma,\ \ \mathcal{T} \vdash t \xrightarrow{r} t' \iff \mathcal{M}(\mathcal{T}) \vdash \mathcal{M}(t \xrightarrow{r} t')$$

implies

$$\forall r \in T_{\Sigma_{\mathit{Time}}}, \forall t, t' \in T_\Sigma,\ \ \mathcal{T} \vdash t \xrightarrow{r} t' \iff \mathcal{T}^t \vdash \delta(r, t) \longrightarrow t'.$$

It can then be proved that

- a system $\widehat{t}$ rewrites to a non-δ-term $\widehat{t'}$ in $\mathcal{T}^t$ iff in TRL t rewrites to t' in $\mathcal{T}$:

$$\forall t, t' \in T_\Sigma,\ \ (\exists r \in T_{\Sigma_{\mathit{Time}}} \mid \mathcal{T} \vdash t \xrightarrow{r} t') \iff \mathcal{T}^t \vdash \widehat{t} \longrightarrow \widehat{t'}$$

- in this translation, no system rewrites to a δ-term:

$$\forall t \in T_\Sigma, \forall t' \in T_{\Sigma^t},\ \mathcal{T}^t \vdash t \longrightarrow t' \text{ implies } t' \in T_\Sigma.$$

The tick rule introduces two variables, x_r and y. This reflects the fact that it is in general undecidable in TRL whether a term rewrites in time r $(r > 0)$, and, even if it is known that t rewrites in time r, it is also in general undecidable whether t rewrites to a given term t' in time r.

6 Concluding Remarks

We have presented a general method for specifying real-time and hybrid systems in rewriting logic, have illustrated it with a well-known benchmark, and have shown how a wide range of real-time and hybrid system models can be naturally expressed in rewriting logic.

The present work should be further extended in several directions, including the following:

- systematic study of the relationships with more abstract levels of specification such as the duration calculus and timed temporal logics;
- study of notions of *distributed real-time* within the rewriting logic formalism;
- further exploration of the use of rewriting strategies in real-time specifications;
- further development of proof techniques and tools, and of symbolic simulation capabilities, within rewriting logic itself;
- development of a good collection of case studies and executable specifications, and further exploration of how other real-time formalisms can be expressed.

Acknowledgments. We cordially thank Saddek Bensalem, Manuel Clavel, Piotr Kosiuczenko, Narciso Martí-Oliet, Sigurd Meldal, Joseph Sifakis, Carolyn Talcott, and Martin Wirsing for their comments and suggestions, that have helped us in the development of these ideas and in improving their presentation.

References

[1] W. M. P. van der Aalst. Interval timed coloured Petri nets and their analysis. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, Springer LNCS 691, pages 453–472, 1993.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[3] Rajeev Alur and David Dill. The theory of timed automata. In J.W. de Bakker, G. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, Springer LNCS 600, 1991.

[4] Michael J. Beeson. *Foundations of Constructive Mathematics.* Springer, 1985.

[5] J. A. Bergstra and J. V. Tucker. Algebraic specification of computable and semicomputable data types. *Theoretical Computer Science*, 50:137–181, 1987.

[6] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. 1996. In this volume.

[7] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[8] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proc., IEEE Real-Time System Symposium, San Juan*, 1994.

[9] P. Kosiuczenko and M. Wirsing. Timed rewriting logic, 1995. Working material for the 1995 Marktoberdorf International Summer School "Logic of Computation".

[10] P. Kosiuczenko and M. Wirsing. Timed rewriting logic for the specification of time-sensitive systems. *Science of Computer Programming*, 1996. To appear.

[11] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-Time: Theory in Practice*, Springer LNCS 600, 1991.

[12] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993.

[13] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[14] J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In *Proc. Concur'96*, Springer LNCS, 1996. To appear.

[15] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[16] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[17] S. Morasca, M. Pezzè, and M. Trubian. Timed high-level nets. *The Journal of Real-Time Systems*, 3:165–189, 1991.

[18] P. C. Ölveczky, P. Kosiuczenko, and M. Wirsing. An object-oriented algebraic steam-boiler control specification. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Application: Specifying and Programming the Steam-Boiler Control.* Springer, 1996. To appear.

[19] Michael Rabin. Computable algebra: General theory and theory of computable fields. *Transactions of the American Mathematical Society*, 95:341–360, 1960.

Electronic Notes in Theoretical Computer Science 4 (1996)

Discrete Event Systems in Rewriting Logic

Christopher Landauer

National Systems Group, The Aerospace Corporation
Herndon, Virginia 20171
E-mail: cal@aero.org

Abstract

In this note, we report on some work in progress on using rewriting logics for discrete event simulation. The idea is to combine the proofs in the logic with the observations in the simulations to gain a better understanding of the interaction intricacies that seem to occur in complex simulations. In particular, we use communication protocols as our application domain, since they have all the interaction and unpredictability that makes formal specifications difficult.

1 Problem: Formal Methods in Simulation

The historical barriers to the use of formal methods in designing and developing communication protocols derive from their different attitudes: verification models have been used for many years for proofs of behavior (a verification model cannot tell you when the model is wrong), but simulation models are used for observations of behavior (a simulation model cannot tell you when the model is right). These are almost always different models, since they must concentrate on different aspects of the behavior.

For example, the verification methods almost always abstract out the notion of time (even temporal logic [16] [17] does not deal with time, only with the possible orderings of events induced by time) and probabilities (it is extremely hard to prove probabilistic statements), whereas the simulations can only show behavior, not explore all possibilities (so mathematical certainty is replaced by statistical certainty).

Our approach here is to implement a discrete event simulation style within rewriting logic [10], so that the logic can be used directly to prove assertions about the simulation. This is part of an effort to apply interesting formal methods [12] [4] in the modeling of communication protocols [6] [7] [8].

[1] This research was supported in part by the Aerospace Corporation's Sponsored Research Program, by the Federal Highway Administration's Office of Advanced Research, and by the Advanced Research Projects Agency's Software and Intelligent Systems Technology Office.

© *1996 Elsevier Science B. V.*

2 Discrete Event Simulation

Here we briefly describe discrete event simulation, to set the stage for our application. We are only concerned with one particular approach to simulation, called the *event-based* approach; many others exist [15] [18] [3] [2]. The main concepts in this approach are event functions, the future events set, scheduling mechanisms, and the use of probabilities. The basic idea is to break down all activity into a collection of atomic events (they need to be atomic only at the time scales of interest for the simulation), and consider each one to be a separate *event function.*

Each event function examines some amount of local state, maybe changes it, and maybe also schedules further events to occur at later times. It has access to whatever global state is in the model and to a certain amount of local state (sometimes, as in our example, the local states are grouped according to different computational entities).

The simulator maintains its own notion of time that is internal to the model, called the *simulation time.* It also maintains a collection of events that have been scheduled but that have not yet occurred, called the *future events set.* "Scheduling events" means adding them to the future events set. If the events occur at sufficiently distant places, then there may be multiple future events sets; of course, in this case, coordination and interference detection are important problems.

Simulation time advances whenever the simulator needs to decide what event should occur next. It takes the event that is to occur "soonest" out of the future events set, sets its notion of the current time from that event, and calls that event function. When there are several events scheduled for the same simulation time, then an arbitration rule is used, the most common of which is to use the event that was scheduled for that time first.

The notion of time is fundamental to simulation. It plays the most important coordinative role in detecting and managing interactions (coincidences of influence), and in reducing the amount of effort the program needs to spend in looking for interactions. Replacing it with distributed notions of time or space requires some kind of cooperative coordination process to account for the loss of this global mechanism.

Finally, since most simulation models are used to study situations that have uncertainties, we have to decide how to model the different kinds of uncertainties. We have a choice of modeling uncertainty with probability or with non-determinism, and since it is very hard to implement non-determinism in a computer, we have historically used "pseudo-random" number generators [13] [1].

2.1 Styles of Simulation

In this section, we describe some different styles of simulation that can be considered. One of the difficulties with almost all simulation systems is that they only support the basic single sample numerical simulation, in which all

the parameters have numerical values, and random samples are chosen as needed. The main power of formal methods here is to examine entire classes of samples without having to elaborate all of them.

One straightforward way to keep track of all the different possibilities is to use expressions of the form

$$p\ S + (1 - p)\ S'$$

with explicit probability numbers p and situations S. This allows the simulation to explore all choices and therefore all traces (we can imagine simplifications that ignore expressions with small enough probabilities). The overall state is a large disjunction, with each term protected by a probability value (the guards' values must add to 1). Each term is a conjunction of state variable expressions that assert current values of state variables. The entire state has a time label, asserting that this is the probability distribution at that (simulation) time.

We can use a probabilistic choice operator $(p?_ : _)$ to express the changes in the probability expressions.

The problem with this approach is combinatorial explosion, since every probabilistic choice leads to an approximate doubling of the size of the expression. We want to do something more interesting.

3 Example: The Send and Wait Protocol

We illustrate our approach with an example. We use the Send-and-Wait Protocol, which is a simple, one message at a time transmission protocol that can account for many typical network services, such as lost, duplicated, and delayed messages. This protocol is related to the Alternating Bit Protocol, which is a kind of *de facto* standard test case for specification languages and methodologies.

Our intention here is to improve the model of [10] in the direction of explicit simulation with numerical observations, without losing the formality that allows proofs of properties. In particular, we will add explicit control of the application of rules corresponding to the different events.

3.1 Send and Wait Protocol Description

The model contains a sender and a receiver, and a channel across which they intend to communicate. The idea is that the sender waits for an acknowledgement from the receiver before it sends the next message; the sender and receiver coordinate which message is being sent with an integer message sequence number, which is assumed to start at zero for both processes (initial coordination is not part of this protocol).

The protocol model has a process that generates new messages, and a channel that transmits the messages in two directions: from the sender to the receiver and the reverse for the acknowledgements. The channel can lose, delay, or duplicate messages.

The sender enqueues generated messages and sends them when it can. It accounts for channel services by sending a message, and waiting for an acknowledgement from the receiver before sending another. The sender is said to be BUSY during this wait time, and IDLE otherwise. If a certain time interval, called the *timeout*, occurs before the sender receives the proper acknowledgement, then it sends the message again and resets the timer.

3.2 Send and Wait Protocol Events

There are six events in this model (the only interesting thing about the data types is that a *packet* includes a message and a sequence number):

- Generate(message msg),
- Timeout(integer sq), and
- RcvAck(packet p) for the sender;
- Rcv(packet p) for the receiver; and
- Send(packet p), and
- Ackn(packet p) for the channel.

The initial event is

- Generate(0),

and it is assumed that the sender and receiver have sequence numbers both initially zero. The sender will use its sequence number to mark the packets it sends, and the receiver will use its sequence number to determine which message it is expecting.

The

- Generate(message msg)

event function has a few simple steps: it first schedules another event

- Generate(new msg) after a certain time interval,

which may be randomly distributed. It appends the message to the sender's pending message queue, and if the sender is not BUSY, it performs a sequence of operations that we call "send the next message": set the sender state to BUSY, take the first message off the pending message queue, make a pending packet from it and the current sender sequence number, and schedule two events

- Send(Pending) after some time for sender computation, and
- Timeout(SndSeqNo) after the timer interval.

The

- Timeout(integer sq)

event function first checks that the sender is BUSY, so that a pending packet is defined, and that the pending packet sequence number is the same as the timeout sequence number (if not, then this is an old timeout that is ignored). Then it simply schedules two events

- Send(Pending) after some time for sender computation, and
- Timeout(SndSeqNo) after the timer interval.

The

- RcvAck(packet p)

event function performs the same check, that the sender is BUSY, so that a pending packet is defined, and that the pending packet sequence number is the same as the sequence number extracted from the acknowledgement packet (if not, then this is an old delayed duplicate acknowledgement packet that is ignored). Then it sets the sender state to IDLE and increments the sender sequence number. Finally, if the sender pending message queue is not empty, it performs the "send the next message" sequence described above.

The

- Rcv(packet p)

event function checks that the received packet sequence number is the same as the receiver's expected sequence number (if not, then this is an old, much delayed duplicate message that is ignored). Then it increments its expected sequence number and schedules an event

- Ackn(p) after some receiver computation time.

The channel provides all the network services that we are trying to alleviate: loss, delay, and duplication of messages. It has two parts that are symmetric: one from the sender to the receiver (the Send() and Rcv() events), and the other from the receiver to the sender (the Ackn() and RcvAck() events). We will only show the first one.

The

- Send(packet p)

event function either loses the packet (in which case it schedules no event), sends it with some delay (in which case it schedules one event

- Rcv(p) after some time for the propagation delay),

or it duplicates the packet (in which case it schedules that event and also a further event

- Rcv(p) after some time for the duplication delay).

The kind of theorem we want to prove about this protocol is that every generated message is eventually delivered, together with some estimates for the expected delay. To that end, we add two global state variables, both of which are lists of messages: one for the messages generated and the other for messages delivered.

4 Discrete Event Simulation in Rewriting Logic

The general rules of our first model of discrete event simulation are as follows. The state is a tuple

(local, global, events),

with "local" being the cartesian product of all the local state spaces, "global" being the global state space, and "events" being the future events set, which is a set of scheduled events

ev@t,

where "ev" is a parameterized event function call, and "t" is the simulation time at which the call is to be made.

These are considered to be linearly ordered by increasing simulation time, so we model it as a list (your implementation may vary: there is a large literature on this data structure, using linked lists, heaps, priority queues, and many more specialized structures [5] [14]). This form of the model insists on complete control over time, as most sequential implementations do. Later on we discuss an alternative model that could be less constrained, but it must still implement a global progress criterion: all events scheduled for times less than t must occur before any event scheduled for a time greater than or equal to t. This condition is difficult to ensure, partly because we do not know what all of the events scheduled for times less than t are for a particular simulation run until time t during that run.

Each event function becomes a collection of conditional rewrite rules that examine their local state, the global state (if any), make some changes, and add new events.

We need a notion of adding a timed event to the future events set, to account for event functions scheduling subsequent events, and a notion of the current simulation time.

4.1 Example in Rewriting Logic

Don't take the syntax too seriously here; it is meant to be both nearly Maude [10] and helpfully plain, not necessarily conforming to any particular implementation (yet).

The sorts are the usual `boolean`, `integer`, and `real` constants, an unspecified `message` type for the message contents (for simulations, we just use integers so we can tell them apart), and some constructed sorts:

```
sort packet = [sn : integer, msg : message],
sort states = {IDLE, BUSY},
sort timed_evt = event @ real.
```

The ops we use include several constants:

```
op IDLE : states,
op BUSY : states, and
op 0 : integer,
```

the component selection functions for packets

```
op _.sn : packet -> integer, and
op _.msg : packet -> message,
```

various logical, arithmetic, and data structure connectives that we will not specify here, and our two application specific operations, representing timed events and adding them to the future events set

```
op _@_ :  event real -> timed_evt, and
op add _ to _ :  timed_evt list -> list.
```

In order to define the add_to_ operation, for adding newly scheduled events to the futures events set, we use two reduction rules:

```
eq add EV@T to [] = [EV@T] .

eq add EV@T to [E@U | L] =
     if (T < U) then [EV@T | [E@U | L]]
     else [E@U | add EV@T to L] .
```

The second rule implements our convention that newly scheduled events go in place *after* previously scheduled events with the same times. This is an irrelevant property for most list insertion sorts, but it is extremely important subtlety for time control in discrete event simulation: if the comparison used <= instead of <, for a more efficient insertion, then the events scheduled for the same time would all be executed as if from a stack instead of a queue. That means that the first event scheduled for a given time could schedule many others to occur at the same time, which means that an arbitrary amount of computation could be done in zero time. With a queue, the first element inserted will be the first one out, regardless of how many are inserted afterwards.

The state is defined by state variables collected into groups with the same intended scope (although we don't enforce that scope in this version of the model). The sender has state variables

```
PState :  states,
SndSeqNo :  integer,
PendMes :  list of message, and
Pending :  packet.
```

The receiver has only the state variable

```
RcvSeqNo :  integer.
```

The global state contains the two message lists:

```
GenMes :  list of message, and
DelMes :  list of message.
```

Now we are ready for the hard part. It is pretty straightforward, as shown in [10], to describe the effect of each kind of event as a `rule`, including (in our case) the scheduling of new events.

To illustrate our example, we describe the effect of three of the events. The `Generate(m)` event is in Figure 1. The function `genint()` computes the inter-generation interval. The `Send(p)` event in Figure 2. the function `uniform()` computes a uniform random number, the function `propdel()` computes the

```
event Generate(m) :
     (global, sender, receiver, [E@U|L]) ==>
          L := add Generate(msg+1)@(U+genint(istime)) to L ;
          GenMes := append(msg, GenMes) ;
          PendMes := append(msg, PendMes) ;
          if (PState != BUSY) then
               {
               PState := BUSY ;
               [msg, PendMes] := PendMes ;
               Pending := [SndSeqNo, msg] ;
               L := add Send(Pending)@(U+sendcomp) to
                    add Timeout(SndSeqNo)@(U+timint) to L ;
               }
     (global, sender, receiver, L) .
```

Fig. 1. Generate(m) event

```
event Send(p) :
     (global, sender, receiver, [E@U|L]) ==>
          if (uniform(0,1) >= errprob) then
               {
               L := add Rcv(p)@(U+propdel(tmdelay)) to
                    if (uniform(0,1) >= duplprob) then
                         add Rcv(p)@(U+dupldel(tmdelay)) to L ;
                    else L ;
               }
     (global, sender, receiver, L) .
```

Fig. 2. Send(p) event

propagation delay, and the function `dupldel()` computes the propagation delay for a duplicate packet. The `Rcv(p)` **event** in Figure 3. In all of them, the sequenced "assignments" are syntactic abbreviations which are composed to compute parts of the entire state tuple. One theorem we would prove here is that `DelMes` is always an initial subsequence of `GenMes`, and another would give an estimate for the delay time between generation and delivery of a message. There is the usual difficult problem of inventing the appropriate induction hypotheses that can be used to derive these and other results.

The interesting part for this approach is that the future events set *determines what rules occur.* Thus, we either need to coalesce all of the rules into one giant case-statement, separated by the event type, or else deal explicitly with the reflection implied by controlling the rule strategy directly from the object language [11]. This is one current focus of our study, and we describe in the next subsection some important issues about the question.

```
event Rcv(p) :
     (global, sender, receiver, [E@U|L]) ==>
          if (p.sn == RcvSeqNo) then
               {
               L := add Ackn(p)@(U+rcvcomp) to L ;
               DelMes := append(msg, DelMes) ;
               RcvSeqNo := RcvSeqNo + 1 ;
               }
     (global, sender, receiver, L) .
```

Fig. 3. Rcv(p) event

4.2 Communication Protocols

The above example is quite different from the one in [10]. It has an explicit, globally coordinated time management, so it is not a concurrent model. However, the above example is not a model of the send and wait protocol, as the one in [10] is; it is a model of a simulation of that protocol. The question is about where the concurrency occurs: explicitly in the model or implicitly in the logic.

Communication protocols form one of the most difficult examples for discrete event simulators, because they involve often large numbers of interacting entities. These large numbers lead to an interest in concurrent or distributed simulation. One of the most difficult parts of implementing distributed simulation is managing global progress. Several complex synchronization methods have been developed for this problem, but there is continuing discussion of better ways to address it, and no consensus has emerged.

The two important kinds of information we want to prove about communication protocols are safety and progress. Safety means that bad things don't happen, and progress means that good things do happen.

Logics are excellent for considering safety properties; they take the form of constraints on conditions in the system (e.g., we never try to resend a message from an empty message list), and our task is to prove that the constraint always holds.

Progress properties are much harder. Typically, we use a temporal logic and try to prove "eventually" properties, or we write simulations, run them many times, and hope that there isn't some unusual situation that causes the system to stop (a deadlock) that we can't find in the simulation.

We must consider progress properties either in the logic, that is, expressed explicitly as expressions in the logic, or of the logic, that is, as explicitly assumed properties of the deductive process itself, in order to get any useful progress results for the protocols. Some kind of fairness was needed for the example in [10], since there was no time observation or management in the example, and even with that fairness, we need a progress assumption about the logic to prove progress of this protocol or any other. Since rewriting logics can be considered to describe a process, we can make our progress properties

in the model depend on progress properties of the logic, something like "if any rewrite rule can be applied, then some applicable rewrite rule will be applied". Our first model makes the connection to progress within the model by arranging that the rewrite rules that can be applied are exactly those for events that should occur "next", and that no pending event disappears.

If we do not want to use the future events set for global coordination, then we have to work harder. We might consider making event selection part of a strategy [11] and leave the ordering of events for the strategy to work out. In the first model, the arbitration rule we used for selecting among events all scheduled for the same time is an example of such a strategy, which was easy to express because all of the pending events were gathered into one place for perusal. In the alternative models, that will no longer be true. Also, Any mechanism that makes a choice of events needs to have some safety constraints, most importantly that no event occurs too early, that is, before another event that is scheduled for an earlier time, in addition to the progress constraint we mentioned earlier.

Any strategy for the alternative models must account for the coordination that the future events set in the first model provides, and we can imagine formal safety constraints on the collection of rules that should be allowed to be enabled according to the pending events. This choice has the advantage of separating the event set management from the rewrite rules for events, which is conceptually nicer, but it does not handle the harder problem: it does not say anything about progress. We still need a notion of progress in the rewrite rules, and also in the guarantees that the strategy mechanisms will be applied.

Any strategy for simulation must have mechanisms for determining when two pending events interfere and when they are completely independent, and for constraining the ordering among the interfering events.

5 Summary and Conclusion

We have described the usual sequential implementation of discrete event simulation scheduling techniques, and shown how it can be implemented rather straightforwardly in rewriting logic, using the Send and Wait protocol as our illustrative example. We have discussed the problems of describing progress in any logic, using communication protocols as our most difficult application example, and shown how progress assertions and global coordination are related.

We believe that rewriting logics have an important role in the difficult problem of combining proofs of concurrent system properties with observing properties of the corresponding discrete event simulations.

References

[1] George S. Fishman, *Monte Carlo: Concepts, Algorithms, and Applications*, Springer-Verlag (1996)

[2] Paul A. Fishwick, *Simulation Model Design and Execution: Building Digital Worlds*, Prentice-Hall (1995)

[3] Paul A. Fishwick, Richard B. Modjeski (eds.), *Knowledge-Based Simulation*, Springer (1991)

[4] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall (1985)

[5] Jeffrey H. Kingston, "Analysis of Henriksen's Algorithm for the Simulation Event Set", *SIAM J. Comput.*, Volume 15, No. 3, pp. 887-902 (August 1986)

[6] Christopher Landauer, "Performance Modeling of Protocols", Paper 16.2, pp. 219-221 in *Proc. MILCOM'84: 1984 IEEE Military Communications Conference*, October 1984, Los Angeles (1984)

[7] Christopher Landauer, "Network and Protocol Modeling Tools", pp. 87-93 in *Proc. 1984 IEEE/NBS Computer Networking Symposium*, December 1984, Gaithersburg, Maryland (1984)

[8] Christopher Landauer, "Communication Network Simulation Tools", Part 3, pp. 995-1001 in *Proc. 16th Annual Pittsburgh Conference on Modeling and Simulation*, April 1985, Pittsburgh, Instrument Society of America (1985)

[9] Christopher Landauer, Kirstie L. Bellman, "Integrated Simulation Environments" (invited paper), *Proceedings of DARPA Variable-Resolution Modeling Conference*, 5-6 May 1992, Herndon, Virginia, Conference Proceedings CF-103-DARPA, published by RAND (March 1993); shortened version in Christopher Landauer, Kirstie Bellman, "Integrated Simulation Environments", *Proceedings of the Artificial Intelligence in Logistics Meeting*, 8-10 March 1993, Williamsburg, Va., American Defense Preparedness Association (1993)

[10] José Meseguer, *A Logical Theory of Concurrent Objects and its Realization in the Maude Language*, Chapter 12, pp. 313-389 in G. Agha, P. Wegner, A. Yonezawa (eds.), *Research Directions in Object-Based Concurrency*, MIT Press (1993)

[11] Manuel Clavel and José Meseguer, *Axiomatizing Reflective Logics and Languages*, pp. 263–288 in: G. Kiczales (ed.), *Proc. Reflection'96*, San Francisco, California (April 1996)

[12] Robin Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer (1980)

[13] Harald Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*, CBMS-NSF Regional Conference Series in Applied Mathematics, Vol. 63, SIAM (1992)

[14] John T. Stasko, Jeffery Scott Vitter, "Pairing Heaps: Experiments and Analysis", *Comm. ACM*, Volume 30, No. 3, pp. 234-249 (March 1987)

[15] Lawrence E. Widman, Kenneth A. Loparo, Norman R. Neilson (eds.), *Artificial Intelligence, Simulation and Modeling*, Wiley (1989)

[16] Pierre Wolper, *Specification and Synthesis of Communicating Processes using an Extended Temporal Logic*, pp. 20-33 in *PoPL 1982: The 1982 ACM Symposium on Principles of Programming Languages* Albuquerque, New Mexico (1982)

[17] Pierre Wolper, "Temporal Logic can be More Expressive", *Information and Control*, Vol. 56, No. 1/2, pp. 72-99 (1983)

[18] Bernard P. Zeigler, *Object-Oriented Simulation with Hierarchical, Modular Models*, Academic Press (1990)

Electronic Notes in Theoretical Computer Science 4 (1996)

A Formal Approach to Object-Oriented Software Engineering

Martin Wirsing and Alexander Knapp [1]

Ludwig-Maximilians-Universität München
Institut für Informatik
Oettingenstraße 67, D-80538 München, Germany
e-mail: {wirsing,knapp}@informatik.uni-muenchen.de

Abstract

The goal of this paper is to show how formal specifications can be integrated into one of the current pragmatic object-oriented software development methods. Jacobson's method OOSE ("Object-Oriented Software-Engineering") is combined with object-oriented algebraic specifications by extending object and interaction diagrams with formal annotations. The specifications are based on Meseguer's Rewriting Logic and are written in an extension of the language Maude by process expressions. As a result any such diagram can be associated with a formal specification, proof obligations ensuring invariant properties can be automatically generated, and the refinement relations between documents on different abstraction levels can be formally stated and proved. Finally, we provide a schematic translation of the specification to Java and thus an automatic generation of an object-oriented implementation.

1 Introduction

Current object-oriented design methods, such as those of Rumbaugh, Shlaer–Mellor, Jacobson and Booch use a combination of diagrammatic notations including object and class diagrams, state transition diagrams and scenarios. Other, academic approaches, such as Reggio's entity algebras, Meseguer's Maude and Ehrich/Sernadas' Troll propose fully formal descriptions for design specifications. Both approaches have their advantages and disadvantages: the informal diagrammatic methods are easier to understand and to apply but they can be ambiguous. Due to the different nature of the employed diagrams and descriptions it is often difficult to get a comprehensive view of all functional and dynamic properties. On the other hand, the formal approaches are more difficult to learn and require mathematical training. But they provide mathematical rigour for analysis and prototyping of designs.

[1] This research has been sponsored by the DFG-project OSIDRIS and the ESPRIT HCM-project MEDICIS.

© *1996 Elsevier Science B. V.*

To close partly this gap we propose a combination of formal specification techniques with pragmatic software engineering methods. Our specification techniques are well-suited to describe distributed object-oriented systems. They are based on Meseguer's Rewriting Logic and are written in an extension of the language Maude. The static and functional part of a software system is described by classical algebraic specifications whereas the dynamic behaviour is modeled by nondeterministic rewriting. The flow of messages is controlled by process expressions.

Jacobson's method OOSE ("Object-Oriented Software-Engineering") is combined with these object-oriented algebraic specifications in such a way that the basic method of Jacobson remains unchanged. As in OOSE the development process of our enhanced fOOSE method consists of five phases: use case analysis, robustness analysis, design, implementation and test. The only difference is that the OOSE diagrams can optionally be refined and annotated by formal text. Any annotated diagram can be semi-automatically translated into a formal specification, i.e. the diagram is automatically translated into an incomplete formal specification which then has to be completed by hand to a formal one.

Thus any fOOSE diagram is accompagnied by a formal specification so that every document has a formal meaning. In many cases the formal specification generates proof obligations which give additional means for validation of the current document. Further proof obligations are generated for the refinement of descriptions, e.g. from analysis to design. These proof obligations can serve as the basis for verification. Finally, due to the choice of the executable specification language Maude early prototyping is possible during analysis and design. Moreover, in many situations we are able to provide a schematic translation of the specification to Java and thus an automatic generation of an object-oriented implementation.

Therefore the combination of algebraic specification with rewriting gives a coherent view of object-oriented design and implementation. Formal specification techniques are complementary to diagrammatic ones. The integration of both leads to an improved design and provides new techniques for prototyping and testing.

Several related approaches are known in the literature concerning the chosen specification formalism and also the integration of pragmatic software engineering methods with formal techniques. First, there is a large body of formal approaches for describing design and requirements of object-oriented systems (for an overview see [9]). Our approach is based on Meseguer's rewriting logic and Maude (cf. e.g. [20]) and was inspired by Astesiano's SMoLCS approach ([3], [23]) which can be characterized as a combination of algebraic specifications with transition systems instead of rewriting. Astesiano was the first integrating also process expressions in his framework. PCF [19] and LOTOS [7] also combine process expressions with algebraic specifications. A process algebra for controlling the flow of messages was introduced in a different way in Maude by [18]. By using an appropriate extension of the μ-calculus Lechner [17] presents a more abstract approach for describing object oriented

requirements and designs on top of Maude. The use of strategies together with rewriting logic was introduced by Vittek et al. [6].

There are also several approaches for integrating pragmatic software engineering methods with formal techniques. Hußmann [13] gives a formal foundation of SSADM, the Syntropy method is based on Z and state charts, Dodani and Rupp [8] enhance the Fusion method by formal specifications written in COLD [15], Lano [16] presents a formal approach to object-oriented software development based on Z++ and VDM++. Very similar to our approach is the one of Futatsugi and Nakajima [22] who use OBJ for giving a formal semantics to interaction diagrams.

The paper is organized as follows: Section 2 gives a short introduction to our chosen specification language Maude and its extension with means for controlling the flow of messages. In section 3 an overview of our enhanced development method fOOSE is presented. Section 4 explains the details of our method for developing a formal specification out of an informal description of a use case and illustrates it by the example of a recycling machine which is the running example of Jacobson's book on OOSE (Object-Oriented Software Engineering, [14]). Section 5 ends with some concluding remarks.

2 Maude

This section gives a short introduction to our chosen specification language Maude (for more details see [21]).

Maude consists of two parts: a purely functional part and an object-oriented part. The functional part is the algebraic specification language OBJ3 [11]; it serves for specifying data types in an algebraic way by equations. The object-oriented part extends OBJ3 by notions of object, message and state, and allows one to describe the dynamic behaviour of objects in an operational style by rewrite rules.

2.1 Functional Part

Maude has two kinds of functional specifications: "modules" and "theories". A module (keyword `fmod ... endfm`) contains an import list (`protecting`, `extending`, or `using`), sorts (`sort`), subsorts (`<`), function (`op`) and variable declarations (`var`), and equations (`eq`) which provide the actual "code" of the module. Theories have different keywords (viz. `fth ... endft`) but have otherwise the same syntax. The real difference is a semantic one: the semantics of a module is the (isomorphism class of the) initial order-sorted algebra [10] whereas a theory is "loose", i.e. it denotes a class of (possibly non-isomorphic) algebras. A module is executable; a theory is not executable, it gives only a few characteristic properties ("requirements") the specified data type has to fulfill.

The following example specifies a trivial theory `TRIV` which introduces one sort `Elt`, and a module `LIST` for the data structure of lists with elements of sort `Elt`. `LIST` is parameterized by `TRIV`.

```
fth TRIV is
  sort Elt .
endft

fmod LIST[X::TRIV] is
  protecting NAT BOOL .
  sort List .
  subsort Elt < List .
  op _ _: List List -> List [assoc id: nil] .
  op length: List -> Nat .
  op _ in _: Elt List -> Bool .
  op _ ≤ _: List List -> Bool .
  var E E': Elt .
  var L L': List .
  eq length(nil) = 0 .
  eq length(E L) = (s 0) + length(L) .
  eq E in nil = false .
  eq E in (E' L) = (E == E') or (E in L) .
  eq (nil ≤ L) = true .
  eq (E L) ≤ (E' L') = (E in (E' L')) and (L ≤ (E' L')) .
endfm
```

For some of the explanations in the following we assume that the reader is familiar with the basic notions of algebraic specifications such as signature, term and algebra (for details see e.g. [24]).

2.2 Object-Oriented Specifications

The object-oriented concept in Maude is the object module. The declaration of an object module (`omod ... endom`) consists, additionally to functional modules, of a number of class declarations (`class`), message declarations (`msg`) and rewrite rules (`rl`).

```
omod BUFFER[X::TRIV] is
  protecting LIST[X] NAT .
  class Buffer | contents: List .
  msg put _ in _: Elt OId -> Msg .
  msg getfrom _ replyto _: OId OId -> Msg .
  msg to _ elt-in _ is _: OId OId Elt -> Msg .
  vars B I: OId .
  var E: Elt .
  var Q: List .
  rl [put] (put E in B) <B: Buffer | contents: Q> =>
            <B: Buffer | contents: E Q>
              if length(Q) < s(s(s(s(s(0))))) .
  rl [get] (getfrom B replyto I) <B: Buffer | contents: Q E> =>
            <B: Buffer | contents: Q>
            (to I elt-in B is E) .
endom
```

An (object) class is declared by an identifier and a list of attributes and their sorts. `OId` is the sort of Maude identifiers reserved for all object identifiers, `CId` is the sort of all class identifiers.

An object is represented by a term—more precisely by a tuple—comprising

a unique object identifier, an identifier for the class the object belongs to and a set of attributes with their values, e.g. `<B: Buffer | contents: X Y Z nil>`.

A message is a term that consists of the message's name, the identifiers of the objects the message is addressed to, and, possibly, parameters (in mixfix notation), e.g. `(put W in B)`.

A Maude program makes computational progress by rewriting its global state, called "configuration" of Maude sort `Configuration` (in the following abstractly denoted by Γ). A configuration is a multiset of objects and messages:

$$\{\!|m_1, \ldots, m_k|\!\} \cup \{\!|o_1, \ldots, o_n|\!\} \quad \text{or, for short} \quad m_1 \ldots m_k \; o_1 \ldots o_n$$

where $\cup$ is a function symbol for multiset union (in Maude denoted by juxtaposition), $m_1, \ldots, m_k$ are messages, and $o_1, \ldots, o_n$ are objects.

A rewrite rule

$$t \xrightarrow{l} t' \Leftarrow \Pi \quad \text{denoted by} \quad [l] \;\; t \;\texttt{=>}\; t' \;\texttt{if}\; \Pi$$

transforms a configuration into a subsequent configuration, where t and t' are terms of sort `Configuration`, Π is a conjunction of equations and l is a label (or proof term) of the form $l(x_1, \ldots, x_k)$ with $x_1, \ldots, x_k$ being the variables occurring in t, t', and Π (we omit these variables). It accepts messages for some objects under a certain condition, possibly modifies these object, and emerges new ones and some additional messages.

In this paper we restrict rewrite rules to those used in Simple Maude of the form

$$m \; o \xrightarrow{l} o' \; o_1 \; \ldots \; o_n \; m_1 \; \ldots \; m_k \Leftarrow \Pi$$

where m, $m_1, \ldots, m_k$ are messages ($k \geq 0$), m being optional, and o, o' (optional), $o_1, \ldots, o_n$ are objects ($n \geq 0$) with o being (possibly) changed to o'.

Formally, we consider a Maude specification M as a quadruple (Σ, E, L, R) given by a signature $\Sigma = (S, F)$, a set E of conditional equations, a set L of labels (also called actions), and a set R of labeled conditional rewrite rules. We assume that for any label there is at most one rule.

Deduction, i.e. rewriting, takes place according to rewriting logic defined by the following four rules (cf. [20]):

(i) Reflexivity.

$$\frac{}{t \xrightarrow{t} t}$$

(ii) Congruence. For each function symbol $f : s_1 \ldots s_n \to s \in F$

$$\frac{t_1 \xrightarrow{\alpha_1} u_1, \; \ldots, \; t_n \xrightarrow{\alpha_n} u_n}{f(t_1, \ldots, t_n) \xrightarrow{f(\alpha_1, \ldots, \alpha_n)} f(u_1, \ldots, u_n)}$$

(iii) Replacement. For each rewrite rule $t_0 \xrightarrow{l} u_0 \Leftarrow \Pi \in R$

$$\frac{t_1 \xrightarrow{\alpha_1} u_1, \; \ldots, \; t_n \xrightarrow{\alpha_n} u_n}{t_0(t_1, \ldots, t_n) \xrightarrow{l(\alpha_1, \ldots, \alpha_n)} u_0(u_1, \ldots, u_n)}, \quad \text{if } \Pi(t_1, \ldots, t_n)$$

(iv) Composition.

$$\frac{t_1 \xrightarrow{\alpha_1} t_2,\ t_2 \xrightarrow{\alpha_2} t_3}{t_1 \xrightarrow{\alpha_1;\alpha_2} t_3}$$

where matching is defined modulo E. (In fact, Maude uses rewriting logic for both its functional and its object-oriented part; we use equational logic for the former one.)

We say that M entails a sequent $t \xrightarrow{\alpha} t'$ if $t \xrightarrow{\alpha} t'$ can be obtained by finite application of the rules above and write $M \vdash t \xrightarrow{\alpha} t'$.

Such a sequent is called one-step concurrent rewrite if it can be derived from R by finite application of the rules (i)–(iv), with at least one application of the replacement rule (iii). It is called a sequential rewrite if it can be derived with exactly one application of (iii).

Since every rewrite step can be decomposed in an (interleaving) sequence of sequential rewrites ([20]) we can restrict our attention to such simple rewrite steps. For any sequential rewrite, we abstract from the actual proof term α and consider only the label l of the unique application of the replacement rule (iii). Moreover, we omit parameters which are not necessary for the synchronisation; mostly this amounts to a statement of the sender and the receiver of a message. A run of M is a possibly infinite chain

$$t_1 \xrightarrow{l_1} t_2 \xrightarrow{l_2} t_3 \xrightarrow{l_3} \dots$$

of one-step sequential rewrites with $M \vdash t_n \xrightarrow{l_n} t_{n+1}$ for every $n \geq 1$.

A (Σ, L)-structure $\mathfrak{A} = ((A_s)_{s\in S}, (f^{\mathfrak{A}})_{f\in F}, (\xrightarrow{l}{}^{\mathfrak{A}})_{l\in L})$ is given by a family $(A_s)_{s\in S}$ of sets, a family $(f^{\mathfrak{A}})_{f\in F}$ of functions with $f^{\mathfrak{A}} : A_{s_1} \times \dots \times A_{s_n} \to A_s$ for $f : s_1 \dots s_n \to s \in F$ and a family $(\xrightarrow{l}{}^{\mathfrak{A}})_{l\in L}$ of relations with $\xrightarrow{l}{}^{\mathfrak{A}} \subseteq A_\Gamma \times A_\Gamma$ for a rewrite rule $t \xrightarrow{l} t' \Leftarrow \Pi \in R$.

$\mathfrak{A}$ is a model of $M = (\Sigma, E, L, R)$ if $\mathfrak{A}$ satisfies all equations of E and all conditional rules R. The semantics of M is defined to be the initial model $\mathfrak{I}$ of all models of M.

A run of $\mathfrak{A}$ is a possibly infinite chain

$$t_1^{\mathfrak{A}} \xrightarrow{l_1}{}^{\mathfrak{A}} t_2^{\mathfrak{A}} \xrightarrow{l_2}{}^{\mathfrak{A}} t_3^{\mathfrak{A}} \xrightarrow{l_3}{}^{\mathfrak{A}} \dots$$

of one-step rewrites.

2.3 Modules with Control

The one-step rewrites build the basis for a small language of processes describing the admissible chains of rewrite steps for particular computations.

Now, an atomic process is a sequential rewrite labeling l. Moreover, there are a constant "1" denoting reflexivity and a constant δ for deadlock which is used to denote a situation where none of the rules below can be applied.

A composite process may be an atomic process, sequential composition, nondeterministic choice, or parallel composition of processes, or a repeat statement. The abstract syntax of processes is given by

$$\mathrm{A} ::= 1 \mid \delta \mid l$$

$$\mathrm{P} ::= \mathrm{A} \mid (\mathrm{P};\mathrm{P}) \mid (\mathrm{P}+\mathrm{P}) \mid (\mathrm{P} \parallel \mathrm{P}) \mid (\mathrm{P}^*)$$

Processes are assumed to satisfy the following laws of Table 1 (borrowed from process algebra PA, see [5]). Note that the last equation for parallel composition induces an interleaving approach to concurrency: either l_1 or l_2 has to be executed first. This assumption simplifies our notion of refinement (cf. 2.4). Any process defines a set of traces it accepts where a trace is a finite or infinite sequence atomic processes.

$$1;p = p, \qquad p;1 = p, \qquad \delta;p = \delta,$$
$$p_1;(p_2;p_3) = (p_1;p_2);p_3,$$

$$\delta + p = p,$$
$$p_1 + p_2 = p_2 + p_1, \qquad p_1 + (p_2 + p_3) = (p_1 + p_2) + p_3,$$
$$(p_1 + p_2);p_3 = p_1;p_3 + p_2;p_3,$$

$$1 \parallel p = p, \qquad \delta \parallel p = \delta,$$
$$p_1 \parallel p_2 = p_2 \parallel p_1, \qquad p_1 \parallel (p_2 \parallel p_3) = (p_1 \parallel p_2) \parallel p_3,$$
$$(p_1 + p_2) \parallel p3 = (p_1 \parallel p_3) + (p_2 \parallel p_3),$$
$$(l_1;p_1) \parallel (l_2;p_2) = l_1;(p_1 \parallel (l_2;p_2)) + l_2;((l_1;p_1) \parallel p_2)$$

$$p^* = (p;p^*) + 1$$

Table 1
Process algebra axioms

With the help of processes we can on the one hand constrain the set of possible runs of a Maude module; on the other hand, processes may trigger certain actions.

To actually incorporate process expressions into Maude specifications, we build up a hierarchy of process definitions $D = ((l_i, p_i))_{1 \le i \le n}$ over a given set of labels L, where $L' = \{l_1, \ldots, l_n\}$ is set of new labels disjoint from L and each process expression p_i uses only labels in $L \cup \bigcup_{1 \le j < i} \{l_j\}$. $L \cup L'$ is called label set of D. Every such hierarchy defines a function $D : L' \to \mathrm{P}$ that maps a new label to a process expression over L; we will also denote its extension to process expressions over $L' \cup L$ by D.

Definition 2.1 A Maude specification with control (M, D, p, q_0) is a Maude specification $M = (\Sigma, E, R, L)$ together with a process definition D over L, a process expression p with labels in the label set of D and an initial configuration $q_0 \in \mathcal{T}(M)_\Gamma$ where $\mathcal{T}(M)_\Gamma$ denotes all terms built from the signature of M of Maude sort `Configuration`.

We say that $(M, D, p, q_0) \vdash t \xrightarrow{\alpha} t'$ with the labels of α in L, if t is reachable from the initial configuration q_0 and if t rewrites to t' via α, i.e. there is an α_0 such that $M \vdash q_0 \xrightarrow{\alpha_0} t$, $M \vdash t \xrightarrow{\alpha} t'$ and any trace of $\alpha_0;\alpha$ is a trace of $D(p)$. Analogously, $(M, D, p, q_0) \vdash t \xrightarrow{s} t'$ for a process expression s with labels in L and that of D, if $(M, D, p, q_0) \vdash t \xrightarrow{\alpha} t'$ for a trace α of $D(s)$. Finally, a run of M is a run of (M, D, p, q_0) if it starts in q_0 and its sequence of labels is a

trace of $D(p)$.

A model $\mathfrak{A}$ of M is a model of (M, D, p, q_0) if every run in (M, D, p, q_0) is a run in $\mathfrak{A}$.

For the concrete syntax, we extend the Maude language by a new keyword `cntrl` to declare the message control that is to be used within `omod ... endom`; since it represents the global control, it is only meaningful in the uppermost module of a hierarchy. The `BUFFER` example could be extended by

```
cntrl [ppput] put(_,B)* .
cntrl [gget] getfrom(B,I); to(I,B,_); getfrom(B,I); to(I,B,_) .
cntrl ppput; gget .
```

The last label-less process declaration defines the global control.

The initial state is not regarded part of a module. It has to provided when opening (starting) a derivation in Maude.

Maude modules that use `cntrl` are called Maude modules with control.

Obviously, every Maude module M is equivalent to a module with control: let $l_1, \ldots, l_n$ be the rule labels of M. Then the process expression $p = (l_1 + \cdots + l_n)^*$ does not restrict the possible runs. Thus M and M extended by `cntrl` p . accept the same runs, if they start with the same configuration.

On the other hand, any Simple Maude module with control can easily be translated to a normal Maude module.

For this purpose, we define two functions hd : P $\rightarrow \wp$(A) and tl : A$\times$P $\rightarrow$ P that compute the accepted atomic processes for an arbitrary process expression and its behaviour after an atomic process has been executed, respectively. These may be easily implemented, since every process expression has an equivalent head normal form (see [4]).

Now, let M be such a module with process definition D, control p and rules $r_1, \ldots, r_n$. First, we flatten p to $D(p)$ by replacing all labels of L' by their corresponding bodies, thus making D superfluous. Next, we construct another module M' which extends M by the import of an implementation of `hd` and `tl` and sorts `A` and `P` for atomic and composed processes (such that `A` is a subsort of `P`). Moreover, M' declares a class `Control` of synchronization objects which have a process expression as attribute:

```
class Control | process: P .
```

Now we define two reductions to Maude, a general one with a global control which works for all process expressions and a more specific one with a distributed control which is well defined only for a set of parallel processes.

In the case of global control, we declare one control object `C0: Control` and initialize it with p. Moreover, for $i = 1, \ldots, n$ we translate any rule

```
rl [r_i] m o => c if Π .      to
rl [r_i] m o <C0: Control | process: Q> =>
  c <C0: Control | process: tl(m,Q)>
```

```
    if m ∈ hd(Q) and Π .
```

Note that M' admits only "interleaving concurrency"; in contrast to M (and thus in contrast to (M, D, p, q_0)) no concurrent rewrite steps are possible in M' among $r_1, \ldots, r_n$.

Note also that using methods as advocated e.g. in [18] M' can further be translated to a module in Simple Maude.

In the case of distributed control we assume that we have k objects $o_1, \ldots, o_k$ and that p is of the form $q_1 \parallel \cdots \parallel q_k$ such that all atomic labels (different from 1 and δ) in q_i denote messages received by o_i $(i = 1, \ldots, k)$. For each o_i we declare a control object c_i: `Control` and initialize it with q_i. Moreover, we translate any rule

```
rl [r] m o_i => c if Π .      to
rl [r] m o_i <c_i: Control | process: Q> =>
  c <c_i: Control | process: tl(m,Q)>
    if m ∈ hd(Q) and Π .
```

Here M' admits "true concurrency": using the replacement rule (iii) of rewriting logic, rules corresponding to different objects can be applied ("fire") concurrently.

Fact 2.2 *Let (M, D, p, q_0) be a module with control, M' its translation to Maude with global control and under the assumptions above M'' its translation to Maude with distributed control. Then (M, D, p, q_0), M', and M'' admit the same runs.*

2.4 Refinement

The principal notion for expressing the correctness of a system wrt. its requirements is the notion of refinement.

Definition 2.3 Let $\mathfrak{A}$ be a (Σ, L)-structure and $\mathfrak{C}$ a (Σ', L')-structure with $\Sigma = (S, F) \subseteq (S', F') = \Sigma'$ and $L \subseteq L'$. Then $\mathfrak{C}$ is a refinement of $\mathfrak{A}$ if there exists a (Σ, L)-substructure $\mathfrak{R} = ((R_s)_{s \in S}, (f^{\mathfrak{R}})_{f \in F}, (\xrightarrow{l}{}^{\mathfrak{R}})_{l \in L})$ of $\mathfrak{C}$ and a Σ-homomorphism $\bar{\cdot} : \mathfrak{R} \to \mathfrak{A}$ which induces a bisimulation, i.e. for any $s \in S$, $r \in R_s$, and $l \in L$ the following holds:

$$\forall a \in A_s : \left((\bar{r} \xrightarrow{l}{}^{\mathfrak{A}} a) \;\Rightarrow\; \exists r' \in R_s : \bar{r'} = a \wedge r \xrightarrow{l}{}^{\mathfrak{R}} r'\right)$$
$$\forall r' \in R_s : \left((r \xrightarrow{l}{}^{\mathfrak{R}} r') \;\Rightarrow\; \exists a \in A_s : \bar{r'} = a \wedge \bar{r} \xrightarrow{l}{}^{\mathfrak{A}} a\right)$$

Let M and M' be two Maude modules (both possibly with control). M' is called a (semantic) refinement of M if the initial model of M' is a refinement of the initial model of M.

This refinement relation is obviously transitive.

The control may be refined in the standard way (see [1] and [2]) by substituting complex process expressions for atomic ones. In our context, a hierarchy

of process definitions is enlarged at the lower end by new process expressions for labels the hierarchy is based on.

Definition 2.4 Let $D = ((l_i, p_i))_{1 \leq i \leq n}$ and $D' = ((l'_i, p'_i))_{1 \leq i \leq n'}$ be process definitions over label sets L and L' respectively, with $L = L_1 \uplus L_2$, $L_1 \cap L' = \emptyset$, and $L_2 \subseteq L'$. Then, D' is called a process definition refinement of D, if it is of the form $((l'_1, p'_1), \ldots, (l'_k, p'_k), (l_1, p_1), \ldots, (l_n, p_n))$ with $\{l'_1, \ldots, l'_k\} \subseteq L_1$.

We distinguish a special refinement relation that will play a major rôle in the sequel.

Definition 2.5 Let $M = ((\Sigma, E, L, R), D, p, q_0)$ and $M' = ((\Sigma', E', L', R'), D', p', q'_0)$ be two Maude specifications with control. We call M' an object control refinement of M if (Σ', E') is a persistent extension of (Σ, E), D' is a process definition refinement of D, $p = p'$, and the following holds:

There is a $C' \subseteq \mathcal{T}(M')_\Gamma$ with $q'_0 \in C'$ and a surjective abstraction function $\tilde{\cdot} : C' \to \mathcal{T}(M)_\Gamma$ of configurations compatible with the equational axioms such that $C'/\tilde{}$ is isomorphic to $\mathcal{T}(M)_\Gamma$ with respect to multiset union (on equivalence classes), $\tilde{q'_0} = q_0$ and

(i) For any sequent $M \vdash c_1 \xrightarrow{l} c_2$ and for any $\tilde{c'_1} = c_1$ there is a corresponding sequent $M' \vdash c'_1 \xrightarrow{l} c'_2$ such that $\tilde{c'_2} = c_2$.

(ii) For any sequent $M' \vdash c'_1 \xrightarrow{l} c'_2$ with $\tilde{c'_1} = c_1$ there is a corresponding sequent $M \vdash c_1 \xrightarrow{l} c_2$ such that $\tilde{c'_2} = c_2$.

Lemma 2.6 *Let M and M' be two Maude modules with control. If M' is an object control refinement of M then M' is a refinement of M.*

Proof. Let $\tilde{\cdot} : \mathcal{T}(M')_\Gamma \supseteq C' \to \mathcal{T}(M)_\Gamma$ be an abstraction function with the required properties. Let $\mathfrak{A}$ be the initial model of M and $\mathfrak{C}$ that of M'. Let $R_\Gamma = \mathfrak{C}(C')$ (where $\mathfrak{C}(C')$ denotes the interpretation of C' in $\mathfrak{C}$), $R_s = C_s$ for any other sort s, and $\bar{\cdot} : \mathfrak{R} \to \mathfrak{A}$ be $\tilde{\cdot}$ transferred to $\mathfrak{R}$ such that $\overline{\mathfrak{R}(r)} = \mathfrak{A}(\tilde{r})$. Then $\mathfrak{R}$ and $\bar{\cdot}$ fulfill the requirements of the refinement definition. □

3 Enhanced OOSE Development Process

The development process of OOSE consists of five phases: use case analysis, robustness analysis, design, implementation and test [14] (see Figure 1).

The use case analysis serves to establish a requirement document which describes the processes of the intended system in textual form. A use case is a sequence of transactions performed by actors (outside the system) and objects (of the system). During the robustness analysis the use cases are refined and the objects are classified in three categories: interaction, control and entity objects. Then in the design phase a system design is derived from the analysis objects and the objects of the reuse library. The design is implemented during the implementation phase and finally during test the implementation is tested with respect to the use case description.

The use case analysis is the particular feature which distinguishes OOSE

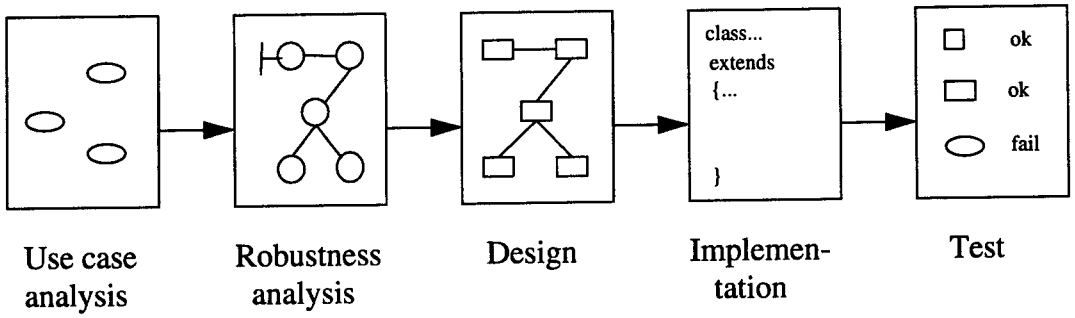


Fig. 1. Development phases of OOSE

from other development methods and which shall be integrated e.g. in the new versions of OMT and Booch's method. Use cases have the advantage to provide a requirement document which is the basis for testing and which can serve as a reference during the whole development.

As in all semiformal approaches one problem is that testing can be done only at a very late stage of development; another problem is the fact that many important requirement and design details can neither be expressed by (the current) diagrams nor well described by informal text.

In our enhanced fOOSE method we provide means to overcome these deficiencies without changing the basic method. The enhanced development process consists of the same phases. The only difference is that the diagrams can optionally be refined and annotated by formal text. Any annotated diagram can be semi-automatically translated into a formal specification, i.e. the diagram is automatically translated into an incomplete formal specification which then has to be completed by hand to a formal one.

Thus any fOOSE diagram is accompagnied by a formal specification so that every document has a formal meaning. In many cases the formal specification generates proof obligations which give additional means for validation of the current document. Further proof obligations are generated for the refinement of descriptions, e.g. from analysis to design. These proof obligations can serve as the basis for verification. Finally, due to the choice of the executable specification language Maude early prototyping is possible during analysis and design. Moreover, in many situations we are able to provide a schematic translation of the specification to Java and thus an automatic generation of an object-oriented implementation.

In the sequel we will use the following method for constructing a formal Maude specification (see Figure 2) from an informal description.

For any given informal description we construct two diagrams: an object model with attributes and invariants, and an enhanced interaction diagram. The object model is used for describing the states of the objects and the (inheritance) relationships, the interaction diagram describes the (data) flow of the messages the objects exchange. The object model directly translates to a specification; the interaction diagram yields an incomplete specification. The

translation of both diagrams yields (after completion) a Maude specification with control together with some proof obligations. Moreover, object control refinement provides the information for tracing the relationship between use case descriptions and the corresponding design and implementation code, the induced proof obligations are the basis for verifying the correctness of designs and implementations.

Further schematic translation to Java provides a direct implementation in an object-oriented language. Our current translation is well-suited to systems composed of a set of concurrently running sequential objects; it might be slow and cumbersome in more complex situations. A further precondition is that the basic data types have efficient implementations. This may not be the case for specifications of the requirements or analysis phase.

Note that in contrast to OOSE we use interaction diagrams also in the analysis phases, and not only in the design phase. We believe that interaction diagrams are good for illustrating the interactions of the objects also at abstract levels.

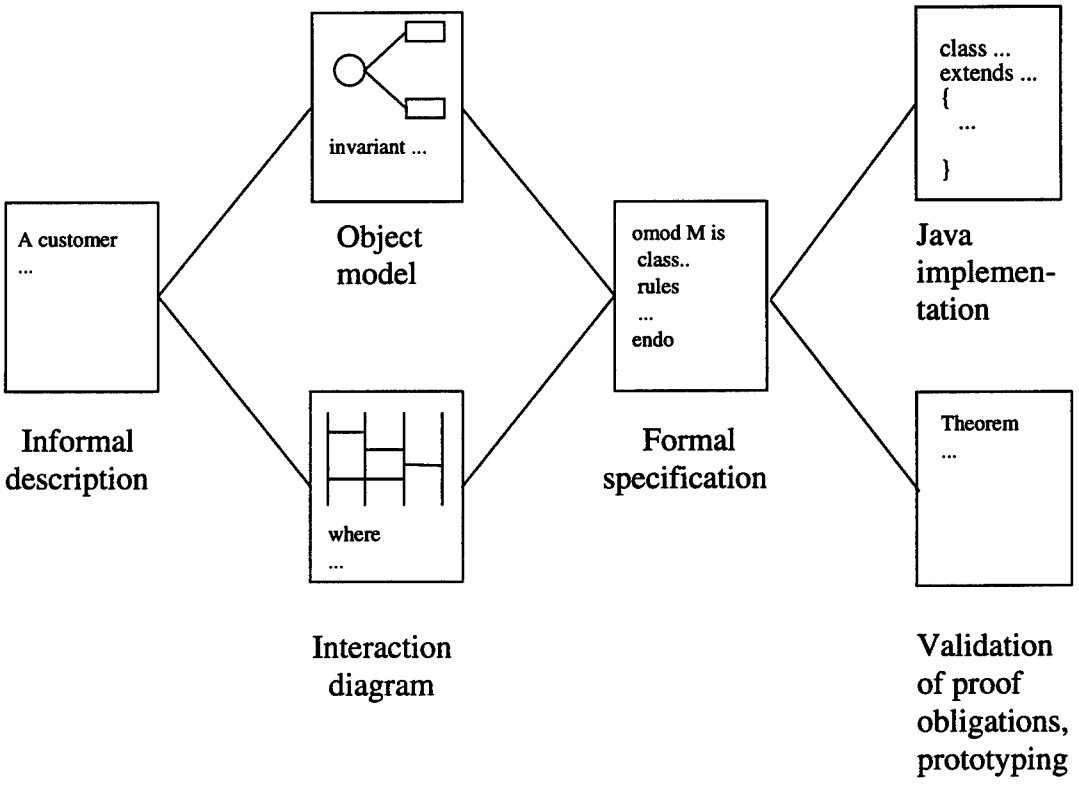


Fig. 2. Construction and use of formal specifications

4 The fOOSE Method in More Detail

In this section we present our method fOOSE (formal Object-Oriented Software Engineering) for developing and refining a formal specification of an informal description of a use case and illustrate it by the example of a re-

cycling machine which is the running example of Jacobson's book on OOSE (Object-Oriented Software Engineering, [14]).

For the construction of a formal specification of a use case we proceed in three steps:

(i) A semi-formal description consisting of an object model and an interaction diagram are developed in the usual OOSE style from the informal (textual) description.

(ii) Functional specifications are constructed for all data types occurring in the diagrams.

(iii) The object diagram is (if necessary) extended by invariants and the interaction diagram is refined. Then a Maude object module is semi-automatically generated from both refined diagrams.

Any refined specification is constructed in the same way. Moreover, for relating the refined "concrete" specification with the more abstract specification one has to give the relationship between the "abstract" and the "concrete" configurations and to define the process definitions for the refined labels. This generates proof obligations (see Section 2.4) which have to be verified to guarantee the correctness of the refinement.

Finally, if the specification is concrete enough, it is schematically translated to a Java program.

We show the specification and refinement activities for the recycling machine example on the level of requirements analysis and robustness analysis in Sections 4.1 and 4.2. In Section 4.3 the generation of the Java code is presented.

4.1 Requirements Analysis

The informal description of the recycling machine consists of three use cases. One of them is the use case "returning items" which can be described in a slightly simplified form as follows:

> "A customer returns several items (such as cans or bottles) to the recycling machine. Descriptions of these items are stored and the daily total of the returned items of all customers is increased. The customer gets a receipt for all items he has returned before. The receipt contains a list of the returned items as well as the total return sum."

We develop a first abstract representation of this use case with the help of an object diagram that describes the objects of the problem together with their attributes and interrelationships, and of an interaction diagram that describes the flow of exchanged messages.

To do this we model the use case as an interactive system consisting of two objects of classes SB and RM (Figure 3 on the left). The class SB stands for "system border" representing the customer, i.e. the actor of this use case. It is modeled without any attributes. The class RM repesents the recycling machine and has two attributes storing the daily total and the current list of

items. For simplicity of presentation both attributes are considered as lists of items.[2] The interaction diagram (Figure 3 on the right) shows (abstractly) the interaction between the customer and the recycling machine. The customer sends a return message containing a list of returned concrete items. The machine prints a receipt with the list of (descriptions of) the returned items as well as the total return sum (in DM). To distinguish between the concrete items and their descriptions in the machine we call the sort of lists of concrete items CList and the other IList.

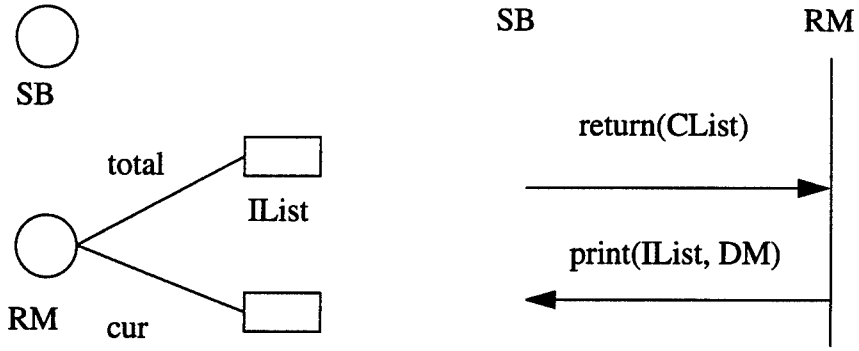


Fig. 3. Object model and interaction diagram of the recycling machine

More generally, an object model consists of several objects (represented by cycles) with their attributes (represented by lines from circles to rectangles) and the relationships between the objects (represented by arrows). Objects are labeled with their class name and attributes with their name (on the line) and the sort of the attribute (below the rectangle). There are several kinds of relationships. In this paper we consider only the inheritance relationship represented by a dotted arrow from the heir to the parent (for an example see Figure 4).

An interaction diagram consists of several objects represented by vertical lines and messages represented by horizontal arrows. Each arrow leads from the sender object to the receiving object. Objects are labeled with the class name and messages with their name and the sorts of their arguments. Progress in time is represented by a time axis from top to bottom: a message below another should be handled later in time. Moreover, an abstract algorithm can be given at the left hand side of the diagram for describing the control flow (for an example see Figure 7).

There are different ways of interpreting interaction diagrams: Jacobson focuses on sequential systems where every message generates a response. Since we aim at asynchronous distributed systems, in our approach we prefer to state the return messages explicitly. Obviously, it would not be difficult to formalize also Jacobson's interpretation.

Object and interaction diagrams give an abstract view of the informal description. But several important relationships are not represented which will

[2] The choice of lists for the daily total here is a premature design decision we make for simplicity of presentation. It would be better to choose an abstract container type.

be expressed by the formal specifications. For example in the use case "return items" there is a connection between the current list and the daily total; moreover, the list of printed items is a description of the list of returned items. The formal specification will be able to express these semantic dependencies. It will also be used to fix the basic data types.

4.1.1 Functional Specifications for Data Types

The functional specifications are written in the functional style of Maude. For any data type occurring in the diagrams a specification is constructed either by reusing predefined modules from a specification library such as `NAT` and `LIST` or designing a completely new specification.

The following specification of items is new. It introduces two sorts `CItem` and `Item` denoting the "concrete" items of the user and the descriptions of these items. The operation `desc` yields the description of any concrete item whereas the operation `price` computes the price whose value will be given in `DM`.

```
fth ITEM is
  protecting DM .
  sorts CItem Item .
  op price: Item -> DM .
  op desc: CItem -> Item .
endft
```

The specification of lists is obtained by instantiating the list module twice, once with concrete items for elements and once with items; in both cases we rename also the sort `List`.

```
make CLIST is LIST[CItem] * (sort List to CList) endmk
make ILIST is LIST[Item] * (sort List to IList) endmk
```

Moreover, we need two more operations: `amount(l)` calculates the sum of the prices of the elements of `l` and `desclist(cl)` converts any "concrete" list `cl` in a list of descriptions.

```
fmod LIST1[I::ITEM] is
  protecting CLIST ILIST .
  op desclist: CList -> IList .
  op amount: IList -> DM .
  var I: Item .
  var Ci: CItem .
  var L: IList .
  var Cl: CList .
  eq desclist(nil) = nil .
  eq desclist(Ci Cl) = desc(Ci) desclist(Cl) .
  eq amount(nil) = 0 .
  eq amount(I L) = price(I) + amount(L) .
endfm
```

4.1.2 Refining The Diagrams

The third step consists of two activities: the extension of the object models by invariants and the refinement of the interaction diagrams.

An invariant is a relation between the attributes of an object or between the objects of a configuration which has to be preserved by all rewriting steps.

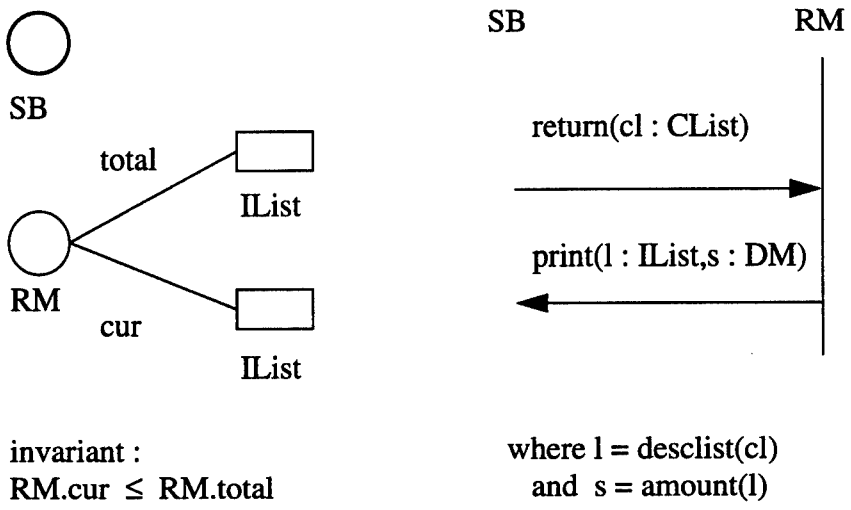


Fig. 4. Object model with invariant and refined interaction diagram of the recycling machine

For example, the attributes total and cur of the recycling machine satisfy the property that all items of cur have also to be in total, i.e. the value of the cur is in the $\leq$ relation (see the specification LIST on Section 2.1) w.r.t the value of total. We express this formally in the object diagram by using a dot notation for selecting the values of the attributes (see Figure 4).

Interaction diagrams are refined in order to express semantic relationships of the parameters of the messages.

We replace the parameter sorts of messages by variables of the appropriate sorts and state the relationships between the variables in an additional "where clause": any message expression $m(s_1, \ldots, s_n)$ is replaced by an expression $m(v_1 : s_1, \ldots, v_n : s_n)$ where $v_1, \ldots, v_n$ are variables of sorts $s_1, \ldots, s_n$; the "where clause" is a conjunction of equations of the form $t_1 = u_1 \wedge \cdots \wedge t_k = u_k$ such that t_j, u_j are terms containing at most the variables $v_1, \ldots, v_n$.

For example, the message expressions return(CList) and print(List, DM) of the interaction diagram in Figure 3 are replaced by return(cl : CList) and print(l : IList, s : DM) where cl, l, and s are variables of sorts CList, List, and DM. Then the equation l = desclist(cl) states that l is a list of descriptions of the elements of cl and the equation s = amount(l) that s is the sum of the prices of l.

The right part of Figure 4 shows the refined interaction diagram.

4.1.3 Construction of a Formal Specification

In this step we show how one can construct semi-automatically a formal specification of the use case from the diagrams. The object model generates the class declarations and invariants; by a combination of the object model with the interaction diagram one can construct automatically a set of (incomplete) rewrite rules which after completion (by hand) define the dynamic behaviour of the use case.

The automatic part of the construction is as follows:

- Every object model induces a set of Maude class declarations:
  - Each object name C with attributes $a_1, \ldots, a_n$ of types $s_1, \ldots, s_n$ of the diagram represents a class declaration

    `class` C `|` a_1`:` s_1`,` ... `,`a_n`:` s_n .

  - Each inheritance relation from D to C corresponds to a subclass declaration

    `subclass` D `<` C .

  - Each invariant I of the attributes $C.a_1, \ldots, C.a_n$ of an object C is translated to the sort constraint

    `sct <O:` `<`C `|` a_1`:` v_1`,` ... `,`a_n`:` v_n`,` a`>:` C
    `if` $I[a_1 \leftarrow v_1] \ldots [a_n \leftarrow v_n]$

    where the constraint condition $I[a_1 \leftarrow v_1] \ldots [a_n \leftarrow v_n]$ is obtained from I by substituting the variables $v_1, \ldots, v_n$ for $C.a_1, \ldots, C.a_n$.
- The interaction diagram induces a set of message declarations:
  - Each message $m(v_1 : s_1, \ldots, v_n : s_n)$ induces the message declaration

    `msg` m`:` `OId` s_1 ... s_n `OId -> Msg` .

    The first argument of m indicates the sender object, the last argument the destination.
- Both diagrams generate the skeleton of a rule:
  - For any message $m(\ldots v_j : s_j \ldots)$ from E_0 to C of the interaction diagram, let

    `class` C `|` ... $a_i : s_i$ ... .
    `class` E_0 `|` ... $b_i : s'_i$ ... .

    be the corresponding class declarations, $m_k(\ldots w_{kj} : s_{kj} \ldots)$ for $1 \leq k \leq n$, be the outgoing messages from C to class E_k of the same activity below m before another message is received by C (if any) and Π the "where clause" of the diagram. Then we obtain the following skeleton of a rewrite

rule:

$$[m]\ m(o_0, \ldots, v_j, \ldots, o)\ \texttt{<}o\texttt{:}\ C\ \texttt{|}\ \ldots\ a_i\texttt{:}\ w_i\ \ldots\ \texttt{>}\ \texttt{=>}$$
$$\texttt{<}o\texttt{:}\ C\ \texttt{|}\ \ldots\ a_i\texttt{:}\ \texttt{?}\ \ldots\ \texttt{>}$$
$$m_1(o, \ldots, w_{1j}, \ldots, o_1)\ \ldots$$
$$m_n(o, \ldots, w_{nj}, \ldots, o_n)$$
$$\texttt{if}\ \Pi\ \texttt{and}\ \Pi\texttt{?}$$

where $o_0, \ldots, o_n$ are object identifiers (for the classes $E_0, \ldots, E_n$).[3]

- The interaction diagram defines a control strategy which is based on the assumption that the objects of the diagram are controlled by (sequential) processes which are composed in parallel:

  For each object the incoming messages are sequentially composed from top to bottom; if a message block is part of a loop, the translated block is surrounded by a repeat statement. These object behaviours are composed in parallel.

- The initial state contains a concrete example of the use case, i.e. the set of objects derived from the object model that are concerned by the interaction diagrams and some messages occuring there.

The rule skeleton expresses that if the object o receives the message m it sends the messages $m_1, \ldots, m_n$. The question marks ? on the right hand side of the rule indicate that the resulting state of o is not expressed in the diagram. Therefore the new values of the attributes have to be added by hand. Similarily, Π? states that the condition is perhaps under-specified.

For example, the diagrams of Figure 4 induce the following skeleton:

```
[ret] return(OO,Items,Rm)
      <Rm: RM | total: W1, cur: W2> =>
        <Rm: RM | total: ?, cur: ?>
        print(Rm,L,S,OO)
          if L = desclist(Items) and S = amount(L) and Π?
```

To get the complete rule one has to fill the question marks with the appropriate value `(l d)` and `l`.

The control strategy interprets the vertical axis as time: the messages have to occur at one object in the defined order. The different objects may act in parallel, controlled by this protocol. The emergence of new messages is left to the object.

In the example, the interaction diagram defines the following control strategy

```
cntrl ret .
```

The full specification of the use case "return items" is as follows:

```
omod RM is
```

[3] Note that in practice only the relevant part of the "where clause" is taken as the condition for the rule, not the full "where clause".

```
protecting LIST1 .
class RM | total: IList, cur: IList .
sct <O: <RM | total: D, cur: L>: RM if L ≤ D .
msg return: OId IList OId -> Msg .
class User .
msg print: OId IList DM OId -> Msg.
var Items: CList .
vars Rm Usr: OId .
vars LO L D: Ilist .
var S: DM .
rl [ret] return(Usr,Items,Rm)
         <Rm: RM | total: D, cur: LO> =>
           <Rm: RM | total: L D, cur: L>
           print(Rm,L,S,Usr)
             if L = desclist(Items) and S = amount(L) .
rl [print] print(Rm,L,S,Usr)
           <Usr: User> =>
             <Usr: User> .
cntrl ret .
endom
```

An invariant I for a class C has to be satisfied by all objects of C and of its subclasses. As a consequence it generates a proof obligation on rewrite rules: every axiom of the form

$$m\ \texttt{<}o\texttt{:}\ Y\ \texttt{|}\ a_1\texttt{:}\ t_1,\ \ldots\ ,a_n\texttt{:}\ t_n,\ a\texttt{>}\ \texttt{=>}$$
$$\texttt{<}o\texttt{:}\ Y\ \texttt{|}\ a_1\texttt{:}\ u_1,\ \ldots\ ,a_n\texttt{:}\ u_n,\ a'\texttt{>}\ c$$
$$\texttt{if}\ \Pi$$

(where Y is C or any of its subclasses) has to satisfy the correctness condition

$$I[a_1 \leftarrow u_1]\ldots[a_n \leftarrow u_n] \Leftarrow \Pi \wedge I[a_1 \leftarrow t_1]\ldots[a_n \leftarrow t_n]$$

For example the rule `[ret]` induces the correctness condition

```
L ≤ (L D) if LO ≤ D and L = desclist(Items)
  and S = amount(L)
```

Obviously, `L ≤ (L D)` holds for any list `L` and `D`. In this case the preconditions are irrelevant.

A possible initial configuration of `RM` can be defined as follows:

$$q_0 = \texttt{<Usr: User> <Rm: RM | total: nil, cur: nil>}$$
$$\texttt{return(Usr,(}ci_1\ ci_2\ ci_3\ \texttt{nil),Rm)}$$

4.2 Robustness Analysis

The use case "return items" is refined in two aspects: instead of returning a list of items the customer returns the items one by one; the machine itself is decomposed into several objects. Accordingly, the informal description consists of a refinement of the use case description of Section 4.1 and a description of the objects of the machine:

"A recycling machine receives returning items (such as cans or bottles) from a customer. Descriptions of these items and the daily total of the returned items of all customers are stored in the machine. If the customer presses the start button he can return the items one by one. If the customer presses the receipt button he gets a receipt for all items he has returned before. The receipt contains a list of the returned items as well as the total return sum."

4.2.1 Object Model With Invariants

To cope with these refinements, in the second phase of OOSE, called "robustness analysis", the objects are classified in three categories: interface, control and entity objects. Interface objects build the interface between the actors (the system border) and the system, the entity objects represent the (storable) data used by the system and the control objects are responsible for the exchange of information between the interface and the entity objects.

Now, the recycling machine consists of five objects (sorts): the interface object Customer_Panel, a control object Receiver and the entity objects Current, Day_Total and Deposit_Item. Customer_Panel and Receiver communicate the data concerning the returned items, the Receiver uses Current and Day_Total for storing and computing the list of current items and the daily total. Deposit_Item stands for all kinds of returned items, in particular for the class of bottles which is modeled as its heir (see Figure 5).

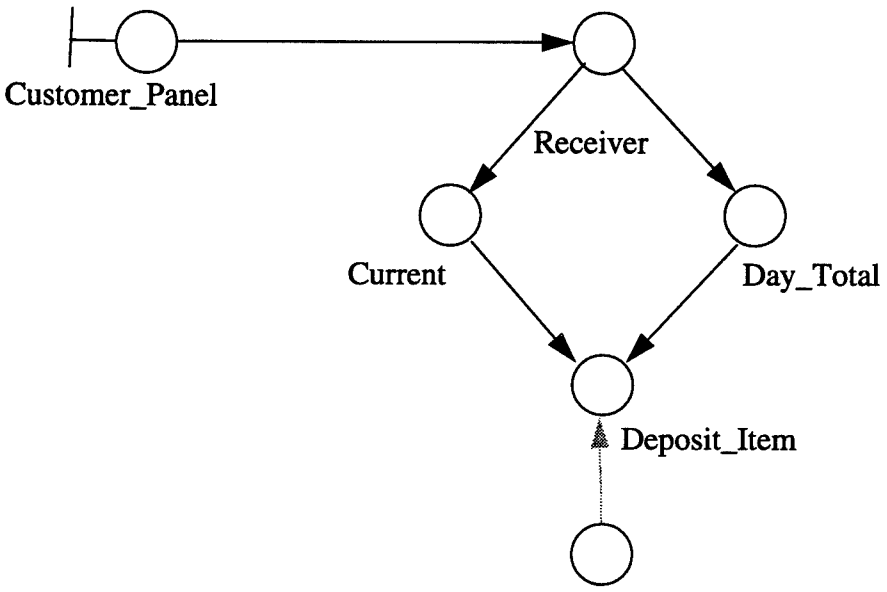


Fig. 5. Object model of the robustness analysis of the recycling machine

In the object model interface objects are represented by hooked circles, control objects by circles with an arrow, and entity objects by full circles.

Additionally, object models are given in two parts, one showing the attributes of the objects and the other showing the relationships between the objects.

In our case, the objects of the robustness analysis have the following attributes (see Figure 6): the Customer_Panel and the Receiver have no attributes; Deposit_item has a name and a price, Bottle has additionally a height and a width; the class Current has a list (of Deposit_item) and an amount as attributes, Day_Total a list of deposit items.

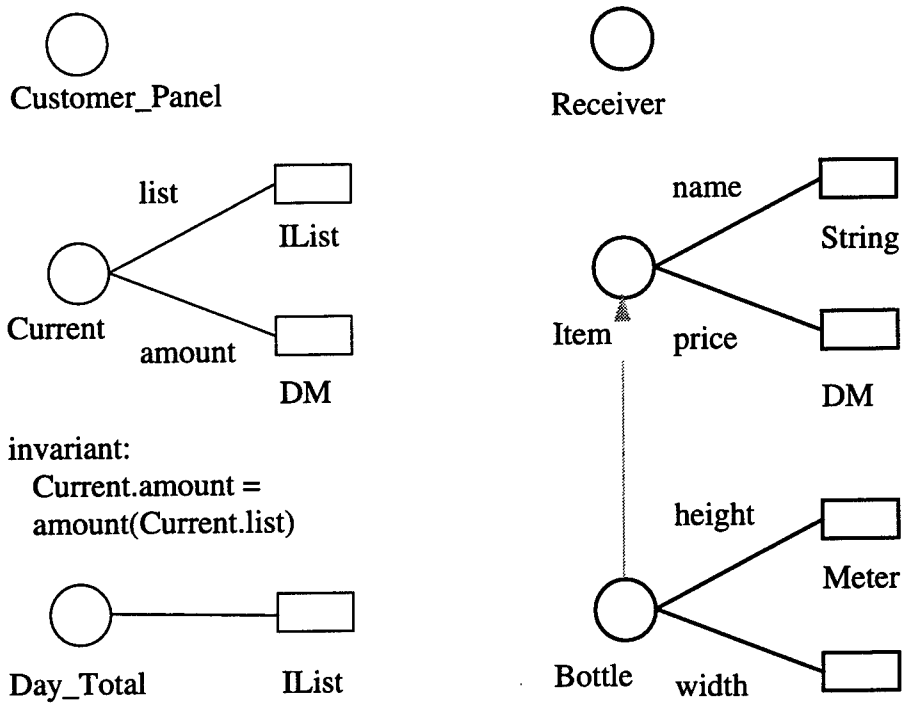


Fig. 6. Object model with attributes and invariants of the robustness analysis of the recycling machine

The attributes of Current satisfy the invariant that the amount is the sum of the prices of the items of the list.

4.2.2 Interaction Diagram

From the informal description one can derive three kinds of messages which are sent from the system border (i.e. from the customer) to the Customer_Panel: a start message, a return message for returning one concrete item and a receipt message for requiring a receipt. Each of these messages begins a new activity of the customer panel. On the other hand, the customer panel sends a print message to the system border.

The start message concerns only the Customer_Panel. After receipt of the return message the customer panel sends a message, say new(i), with the description i of the concrete item to the receiver. Then the receiver forwards this information to Current and Day_Total by two messages, both being called add; the end of such a return process is to be acknowledged by a message ack. In the third activity the Customer_Panel sends a print_receipt request to the Receiver which in turn sends a standard get message to Current. After getting the answer the Receiver forwards the answer to the Customer_Panel (by a message called send) which prints the result.

The resulting interaction diagram derived from this text is, in the next step, refined by inserting variables for the parameters of messages and by stating semantic properties of the parameters (see Figure 7).

In particular the diagram shows that the description i of a returned item ci is not changed and that the amount of the print message is compatible with the prices of the returned items.

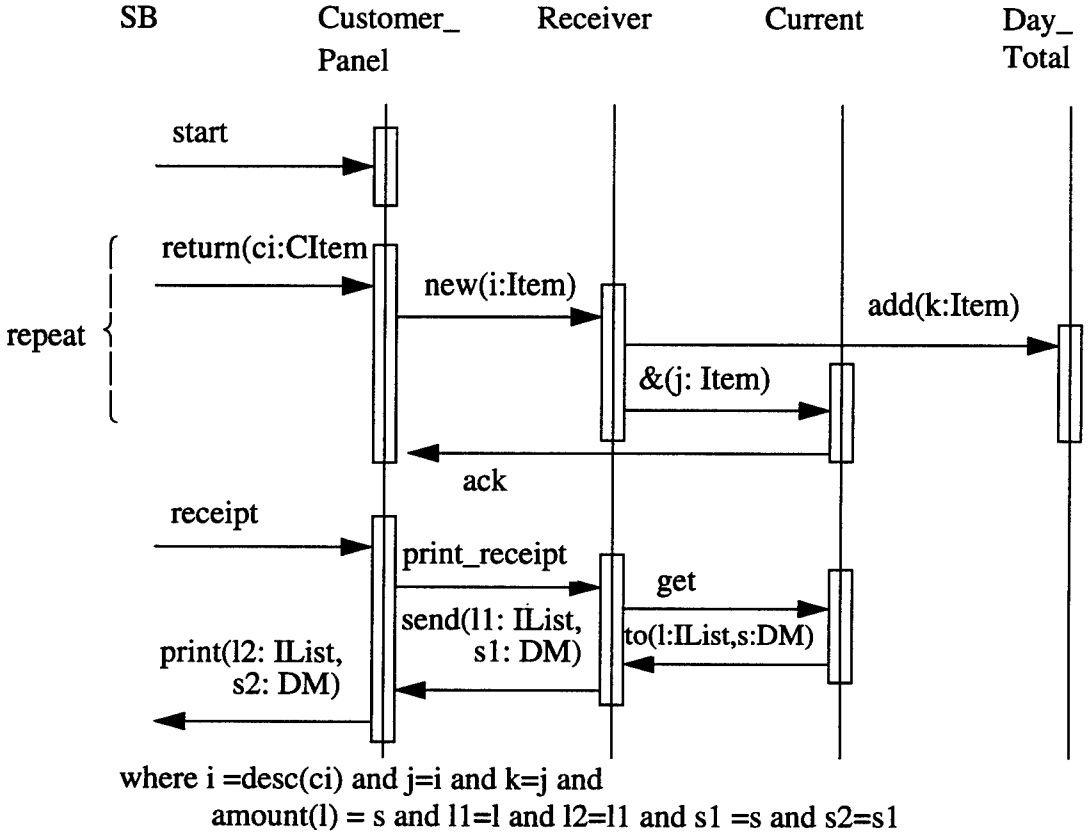


Fig. 7. Refined interaction diagram of the robustness analysis of the recycling machine

4.2.3 Construction of a Formal Specification

The refined diagram generates automatically eleven message declarations according to our method in Section 4.1.3, e.g.

```
msg start: OId OId -> Msg .
msg return: OId CItem OId -> Msg .
msg new: OId Item OId -> Msg .
```

To define the rule skeletons for the interaction diagram of the recycling machine we use the attributes defined in the object model (Figure 5). Then we get the following skeletons for e.g. the start, return and new message:

```
[start] start(Sb,Cp)
        <Cp: Customer_Panel | state: A> =>
          <Cp: Customer_Panel | state: ?>
            if ΠΙ?
```

```
[return] return(Sb,Ci,Cp)
         <Cp: Customer_Panel | state: A> =>
           <Cp: Customer_Panel | state: ?>
           new(Cp,I,Rc)
             if I = desc(Ci) and Π?
[new] new(Cp,I,Rc)
      <Rc: Receiver> =>
        <Rc: Receiver>
        &(Rc,I,Cur)
        add(Rc,I,Dt)
          if Π?
```

The behaviour of the interaction diagram is represented by the following strategy:

```
  (start; (return)*; ack; receipt; send)
|| ((new)*; print_receipt; to)
|| ((&)*; get)
|| (add)*
```

To get the full rules one has to add the state changes and the necessary preconditions. We require preconditions only for the behaviour of the customer panel: pressing the start button should actually start the machine only if it is in state off, returning an item and requiring a receipt should be possible only if the machine is on. By filling in values also for the other question marks we obtain the following rules:

```
rl [start] start(Sb,Cp)
           <Cp: Customer_Panel | state: A> =>
             <cp: Customer_Panel | state: on>
               if A = off .
rl [return] return(Sb,Ci,Cp)
            <Cp: Customer_Panel | state: A> =>
              <Cp: Customer_Panel | state: on>
              new(Cp,I,Rc)
                if I = desc(Ci) and A = on .
rl [new] new(Cp,I,Rc)
         <Rc: Receiver> =>
           <Rc: Receiver>
           &(Rc,I,Cur)
           add(Rc,I,Dt) .
```

Moreover, the following proof obligation is automatically created:

```
price(I)+S = amount(I L) if S = amount(L)
```

(whose proof follows trivially from the definition of amount).

A possible initial configuration of this specification can be defined as follows:

```
q0 = <Usr: User>
       <Cp: Customer_Panel | state: off>
       <Rc: Receiver>
       <Cur: Current | list: nil, amount: 0>
       <Dt: Day_Total | list: nil>
```

```
start(Sb,Cp)
return(Sb,ci1,Cp) return(Sb,ci2,Cp) return(Sb,ci3,Cp)
receipt(Sb,Cp)
```

The text of the full specification can be found in Appendix A.

It remains to prove, that the specification of the robustness analysis step is a refinement of the specification of the requirements analysis. Actually, we will prove that it is an object control refinement: The former class RM is now represented by the four classes Customer_Panel, Receiver, Current, and Day_Total, i.e. each instance of RM is replaced by one instance of the mentioned classes each. More precisely, we set

```
<Cp: Customer_Panel | state: A>
  <Rc: Receiver>
  <Cur: Current | list: L, amount: S>
  <Dt: Day_Total | list: L> =
<Rm: RM | total: Dt.L, cur: Cur.L>
```

The control expression is trivially refined by

```
cntrl [ret]   (start; (return; ack)*; receipt; send)
               || ((new)*; print_receipt; to)
               || ((&)*; get)
               || (add)*
```

Fact 4.1 *The robustness analysis specification of this section is an object control refinement of the requirements specification of Section 4.1.*

4.3 Design and Implementation

In the design step, the analysis model of the system is transformed and refined in the light of the actual implementation environment. In our case, this will be the programming language Java with its extensive class libraries ([12]).

In general, there are three ways to proceed: First, the robustness analysis model can be directly implemented using ad-hoc-methods, but guided by some heuristics; that would amount to the original OOSE method. Second, the Maude module with control that was the result of the specification process can be implemented, either by using the translation with global control of Section 2.3 or by re-implementing the rewriting and the control mechanism. Third, one can make use of the special simple structure of this resulting specification which satisfies the assumptions of our translation with local control (in Section 2.3). Moreover, since the behaviour of any object of the interaction diagram is sequential the computations of hd and tl are particularly simple and can be represented by a finite automaton. Thus we can consider an interaction diagram as a set of asynchronously running concurrent automata which run in parallel to the method calls defined by the rewrite rules.

We will follow this third option. However, it seems worth trying to extend steadily the transformable constructs of Maude in order to make the analysis–

design-step more natural.

In our Java implementation, every object is provided with control, organized as a finite state machine. This makes use of the fact, that inside an object there is only sequentiality. The only branching states of the controlling automaton are loop starting points where a decision is to be taken whether the body of the loop is entered or not. (For this decision we must require that the first action inside the loop and the first action after the loop are different.)

Every method checks if the object is in a state to accept the message called. If this is not the case the call is refused. The sending object—which is obliged to make this specific call—has to wait for a state change of the object called.

Now, a Maude module with control that is the result of the specification process shown is implemented in Java as follows:

- The underlying equational theory is translated to suitable Java functions. (We omit this translation.)
- Every class defines a separate Java class; all attributes of the Maude class are taken over by the Java class.
- Every message to a class, i.e. every message on the left hand side of a rule that occurs together with that class, defines an instance method of the corresponding Java class. Only the formal parameters that do not concern the sending and the accepting class are taken over.
- The control part of the Maude module defines an automaton for every object: Each class extends a special `Foose` class that itself extends the Java class `Thread`.

```
class Foose extends Thread
{
  private protected int acc,got;
  private protected Object[] env;

  private protected void notifyenv()
  {
    for (int i=0;i<env.length;i++)
    { if (env[i]!=this)
      { synchronized (env[i])
        { env[i].notify(); } } }
  }

  private protected void accept(int s)
  {
    synchronized (this)
    { got=0; acc=s; }
    notifyenv();
    while ((got&acc)==0)
    { Thread.yield(); }
    acc=0;
```

```
  }
}
```

This `Foose` class provides the attributes and methods necessary to implement the control automaton: There is an attribute `acc` which represents the state. If a message was accepted this is stored in another attribute `got`. A method `accept` serves for the manipulation of the state and the acknowledgement of the other participating objects (in `env`) of a change in state.

The state itself is coded as a disjunction of the allowed messages. For this purpose, additionally, each class contains class constants M for the possible messages m. Finally, the `run()`-method of `Thread` is re-defined by an implementation of the control-automaton using the `accept()`-method.

The automaton is constructed in the standard way by calculating (using for example hd and tl) the accepted traces of the control part that belongs to the object. For the repeat statement a `while`-loop is constructed that stops if one of the possible first messages after the repeat statement has arrived; the body of the while loop must—because its first message will have arrived before it is executed—accept the messages of the control expression in the order rotated one to the left.

- Every rule defines (different parts of) a body of an Java instance method. The translation is performed in a natural way which we omit. Merely the methods are enriched by synchronization code:

```
public boolean m()
{
  if (M&acc!=M)
    return false;
  else
  { synchronized (this)
    { got=M;
      notifyenv(); }
    ...
    return true; }
}
```

A method call m of another object o is replaced by

```
while (!o.m( ... ))
{ synchronized (this)
  { try { wait(); }
    catch (InterruptedException ignored) { } } }
```

Note, that the automaton in the `run()`-method largely corresponds to the distributed control expressions of the Maude implementation (see Section 2.3).

For the recycling machine this means for example:

```
class Receiver extends Foose
{
  private final static int NEW=1, PRINTRECEIPT=2, TO=4;
  private CustomerPanel cp;
  private Current cur;
  private DayTotal dt;

  public Receiver()
  {
    System.err.println("Receiver");
  }

  public void reg(CustomerPanel c,Current u,
                  DayTotal d,Object[] e)
  {
    cp=c; cur=u; dt=d; env=e;
  }

  public boolean mynew(Item i)
  {
    if ((NEW&acc)!=NEW)
      return false;
    else
    { System.err.println("new()");
      synchronized (this)
      { got=NEW;
        notifyenv(); }
      while (!dt.add(i))
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      while (!cur.conc(i))
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      return true; }
  }

  public boolean printreceipt()
  {
    if ((PRINTRECEIPT&acc)!=PRINTRECEIPT)
      return false;
    else
    { System.err.println("printreceipt()");
      synchronized (this)
      { got=PRINTRECEIPT;
        notifyenv(); }
      while (!cur.get())
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      return true; }
  }

  public boolean to(IList l,int s)
  {
```

```
    if ((TO&acc)!=TO)
      return false;
    else
    { System.err.println("to()");
      synchronized (this)
      { got=TO;
        notifyenv(); }
      while (!cp.send(1,s))
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      return true; }
  }
  public void run()
  {
    System.err.println("runrc");
    while ((got&PRINTRECEIPT)!=PRINTRECEIPT)
      accept(PRINTRECEIPT|NEW);
    accept(TO);
  }
}
```

The complete Javs program and a test run of the initial configuration can be found in Appendix B. It runs with the so-called "appletviewer" program.

5 Concluding Remarks

In this paper we have presented an extension of OOSE by formal specifications which has several advantages:

- The formal meaning of diagrams provides possibilities for prototyping and generates systematically proof obligations for which can serve for validation activities.
- The refinement relation gives the information for tracing the relationships between use case descriptions and the corresponding design and implementation code, the generated proof obligations form the basis for the verification of the correctness of designs and implementations.
- The operational nature of our specification formalism allows one to generate directly Java code from design specifications.
- Traditional OOSE development can be used in parallel with fOOSE since all OOSE diagrams and development steps are valid in fOOSE.

However, there remain several open problems and issues. Our formal annotations of the interaction diagrams cover only repeat statements, means for if and while statements should be added as well. Interaction diagrams as in Jacocson's OOSE are inherently sequential, the whole OOSE method is designed for the development of sequential systems. In contrast to this we focus on the description of distributed concurrent systems. Therefore we need also means for describing the concurrent behaviour in our diagrams, not only in the interaction diagrams but also in other kinds such as use case diagrams.

Another problem is that our notion of refinement is defined on the level of specifications. For software engineers it would be easier if we could define also a refinement relation on the level of diagrams which ensures the validity of an object control refinement.

Finally, our Java implementation has two drawbacks: until now we do not have any formal semantics of Java which makes it impossible to prove the correctness of our translation to Java. To compensate this we plan to define a rewriting logic semantics of central parts of Java which would allow us to perform correctness proofs. The second drawback concerns the style of our implementation which uses heavily the "synchronization code" of the process expressions. In many cases this code is superfluous since the control is already induced by the "natural" data flow of the messages (e.g. in the recycling machine example it would be enough to construct an automaton for the customer panel; all other synchronization code could be omitted). We are investigating data flow analysis methods for eliminating unnecessary synchronizations.

References

[1] L. Aceto and M. Hennessy. Towards action-refinement in process algebras. *Inf. Comp.*, 103:204–269, 1993.

[2] L. Aceto and M. Hennessy. Adding action-refinement to a finite process algebra. *Inf. Comp.*, 115:179–247, 1994.

[3] E. Astesiano, G. Mascari, G. Reggio, and M. Wirsing. On the parameterized algebraic specification of concurrent processes. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *TAPSOFT'85, Vol. 1*, volume 185 of *LNCS*, pages 342–358, Berlin, 1985. Springer.

[4] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge University Press, Cambridge, 1990.

[5] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theo. Comp. Sci.*, 37:77–121, 1985.

[6] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. This volume.

[7] E. Brinksma, editor. LOTOS: A formal description technique based on the temporal ordering of observational behaviour. Technical Report Dis 8807, ISO, 1987.

[8] M. Dodani and R. Rupp. Integrating formal methos with object-oriented methodologies. In M. Wirsing, editor, *ICSE–17 Workshop on Formal Methods Applications in Software Engineering Practice*, Seattle, 1995.

[9] H. D. Ehrich, M. Gogolla, and A. Sernadas. Objects and their specification. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, volume 655 of *LNCS*, pages 40–65, Berlin, 1993. Springer.

[10] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theo. Comp. Sci.*, 105:217–273, 1992.

[11] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ3. Technical Report SRI-CSL-92-03, SRI, 1992.

[12] J. Gosling and H. McGilton. *The Java Language Environment: A White Paper.* Sun Microsystems, Mountain View, Oct. 1995.

[13] H. Hußmann. Formal foundations for pragmatic software engineering methods. In B. Wolfinger, editor, *Innovationen bei Rechnern und Kommunikationssystemen*, pages 1–50, Berlin, 1994. Springer.

[14] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering.* Addison–Wesley, Wokingham, England, 4th edition, 1993.

[15] H. B. M. Jonkers. An introduction to cold-k. In J. A. B. M. Wirsing, editor, *Algebraic methods: theory, tools and applications*, volume 394 of *LNCS*, pages 139–206, Berlin, 1989. Springer.

[16] K. Lano. *Formal Object-Oriented Development.* Springer, London, 1995.

[17] U. Lechner. Object-oriented specifications of distributed systems in the μ-calculus and Maude. This volume.

[18] U. Lechner, C. Lengauer, F. Nickl, and M. Wirsing. (Objects + Concurrency) & Reusability — A Proposal to Circumvent the Inheritance Anomaly. In *Proc. Europ. Conf. Object-Oriented Programming '93*, LNCS, Berlin, 1996. Springer. To appear.

[19] S. Mauw. An algebraic specification of process algebra. In J. A. B. M. Wirsing, editor, *Algebraic methods: theory, tools and applications*, volume 394 of *LNCS*, Berlin, 1989. Springer.

[20] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–389. MIT Press, Cambridge, Massachusetts–London, 1991.

[21] J. Meseguer and T. Winkler. Parallel programming in Maude. In J. Banatre and D. le Metayer, editors, *Research Directions in High-Level Parallel Languages*, volume 574 of *LNCS*, pages 253–293, Berlin, 1992. Springer.

[22] S. Nakajima and K. Futatsugi. Constructing OBJ specifications with object-oriented design methodoly, 1996. To appear.

[23] G. Reggio. Entities: an institution for dynamic systems. In H. Ehrig, K. P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, volume 534 of *LNCS*, pages 244–265, Berlin, 1991. Springer.

[24] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, pages 675–788. Elsevier, Amsterdam, 1990.

A Complete Maude Specification

```
fth STATE is
  sort State .
  op on: -> State .
  op off: -> State .
  op wait: -> State .
endft

fth DMM is
  protecting NAT .
  sort DM < NAT .
endft

omod RM is
  protecting DMM, STATE, CLIST, ILIST .

  class Usr .
  class Customer_Panel | state: State .
  class Receiver .
  class Current | list : IList, amount : DM .
  class Day_Total | list : IList .

  msg start: OId OId -> Msg .
  msg return: OId CItem OId -> Msg .
  msg new: OId Item OId -> Msg .
  msg add: OId Item OId .
  msg &: OId Item OId .
  msg ack: OId OId .
  msg receipt: OId OId .
  msg print_receipt: OId OId .
  msg get: OId OId .
  msg to: OId IList DM OId .
  msg send: OId IList DM OId .
  msg print: OId IList DM OId .

  vars Sb,Cp,Rc,Cur,Dt: OId .
  var A: State .
  var CI: CItem .
  var I: Item .
  var L: IList .
  var S: DM .

  rl [start] start(Sb,Cp)
             <Cp: Customer_Panel | state: A> =>
               <Cp: Customer_Panel | state: on>
                 if A = off .
  rl [return] return(Sb,Ci,Cp)
              <Cp: Customer_Panel | state: A> =>
                <Cp: Customer_Panel | state: on>
                new(Cp,I,Rc)
                  if I = desc(Ci) and A = on .
  rl [new] new(Cp,I,Rc)
           <Rc: Receiver> =>
             <Rc: Receiver>
             &(Rc,I,Cur)
             add(Rc,I,Dt) .
  rl [&] &(Rc,I,Cur)
```

```
          <Cur: Current | list: L, amount: S> =>
            <Cur: Current | list: I L,
                            amount: price(I)+S> .
            ack(Cur,Cp) .
  rl [add] add(Rr,I,Dt)
           <Dt: Day_Total | list: L> =>
             <Dt: Day_Total | list: I L> .
  rl [ack] ack(Cur,Cp)
           <Cp: Customer_Panel> =>
             <Cp: Customer_Panel>
  rl [receipt] receipt(Sb,Cp)
               <Cp: Customer_Panel | state: A> =>
                 <Cp: Customer_Panel | state: wait>
                 print_receipt(Cp,Rc)
                   if A = on .
  rl [print_receipt] print_receipt(Cp,Rc)
                     <Rc: Receiver> =>
                       <Rc: Receiver>
                       get(Rc,Cur) .
  rl [get] get(Rc,Cur)
           <Cur: Current | list: L, amount: S> =>
             <Cur: Current | list: nil, amount: 0>
             to(Cur,L,S,Rc) .
  rl [to] to(Cur,L,S,Rc)
          <Rc: Receiver> =>
            <Rc: Receiver>
            send(Rc,L,S,Cp) .
  rl [send] send(Rc,L,S,C)
            <Cp: Customer_Panel | state: A> =>
              <Cp: Customer_Panel | state: off>
              print(CP,L,S,SB)
                if A = wait .
endom
```

B Complete Java Program

```
class Foose extends Thread
{
  private protected int acc,got;
  private protected Object[] env;

  private protected void notifyenv()
  {
    for (int i=0;i<env.length;i++)
    { if (env[i]!=this)
      { synchronized (env[i])
        { env[i].notify(); } } }
  }

  private protected void accept(int s)
  {
    synchronized (this)
    { got=0; acc=s; }
    notifyenv();
    while ((got&acc)==0)
```

```
    { Thread.yield(); }
    acc=0;
  }
}

class CustomerPanel extends Foose
{
  private final static int START=1, RETURN=2, ACK=4,
                           RECEIPT=8, SEND=16;
  private Receiver rc;
  private State state;

  public CustomerPanel()
  {
    System.err.println("CustomerPanel");
  }

  public void reg(Receiver r,Object[] e)
  {
    rc=r; env=e;
  }

  public boolean mystart()
  {
    if ((START&acc)!=START)
      return false;
    else
    { if (state==OFF)
      { System.err.println("start()");
        synchronized (this)
        { got=START;
          notifyenv(); }
        state=ON;
        return true; }
      else
        return false; }
  }

  public boolean myreturn(CItem ci)
  {
    if ((RETURN&acc)!=RETURN)
      return false;
    else
    { if (state==ON)
      { Item i=ci.desc();
        System.err.println("return()");
        synchronized (this)
        { got=RETURN;
          notifyenv(); }
        while (!rc.mynew(i))
        { synchronized (this)
          { try { wait(); }
            catch (InterruptedException ignored) { } } }
        return true; }
      else
        return false; }
  }
```

```
public boolean ack()
{
  if ((ACK&acc)!=ACK)
    return false;
  else
  { System.err.println("ack()");
    synchronized (this)
    { got=ACK;
      notifyenv(); }
    return true; }
}

public boolean receipt()
{
  if ((RECEIPT&acc)!=RECEIPT)
    return false;
  else
  { if (state==ON)
    { System.err.println("receipt()");
      synchronized (this)
      { got=RECEIPT;
        notifyenv(); }
      state=WAIT;
      while (!rc.printreceipt())
        synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } }
      return true; }
    else
      return false; }
}

public boolean send(IList l,int s)
{
  if ((SEND&acc)!=SEND)
    return false;
  else
  { if (state==WAIT)
    { System.err.println("send()");
      synchronized (this)
      { got=SEND;
        notifyenv(); }
      state=OFF;
      System.out.println(l);
      System.out.println(s);
      return true; }
    else
      return false; }
}

public void run()
{
  System.err.println("runcp");
  accept(START);
  while (true)
  { accept(RETURN|RECEIPT);
```

```
      if ((got&RECEIPT)==RECEIPT)
        break;
      accept(ACK); }
    accept(SEND);
  }
}

class Receiver extends Foose
{
  private final static int NEW=1, PRINTRECEIPT=2, TO=4;
  private CustomerPanel cp;
  private Current cur;
  private DayTotal dt;

  public Receiver()
  {
    System.err.println("Receiver");
  }

  public void reg(CustomerPanel c,Current u,
                  DayTotal d,Object[] e)
  {
    cp=c; cur=u; dt=d; env=e;
  }

  public boolean mynew(Item i)
  {
    if ((NEW&acc)!=NEW)
      return false;
    else
    { System.err.println("new()");
      synchronized (this)
      { got=NEW;
        notifyenv(); }
      while (!dt.add(i))
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      while (!cur.conc(i))
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      return true; }
  }

  public boolean printreceipt()
  {
    if ((PRINTRECEIPT&acc)!=PRINTRECEIPT)
      return false;
    else
    { System.err.println("printreceipt()");
      synchronized (this)
      { got=PRINTRECEIPT;
        notifyenv(); }
      while (!cur.get())
      { synchronized (this)
        { try { wait(); }
```

```
            catch (InterruptedException ignored) { } } }
        return true; }
    }

    public boolean to(IList l,int s)
    {
      if ((TO&acc)!=TO)
        return false;
      else
      { System.err.println("to()");
        synchronized (this)
        { got=TO;
          notifyenv(); }
        while (!cp.send(l,s))
        { synchronized (this)
          { try { wait(); }
            catch (InterruptedException ignored) { } } }
        return true; }
    }

    public void run()
    {
      System.err.println("runrc");
      while ((got&PRINTRECEIPT)!=PRINTRECEIPT)
        accept(PRINTRECEIPT|NEW);
      accept(TO);
    }
}

class Current extends Foose
{
    private final static int CONC=1, GET=2;
    private CustomerPanel cp;
    private Receiver rc;
    private IList list;
    private int amount;

    public Current()
    {
      System.err.println("Current");
    }

    public void reg(CustomerPanel c,Receiver r,Object[] e)
    {
      cp=c; rc=r; env=e;
    }

    public boolean conc(Item i)
    {
      if ((CONC&acc)!=CONC)
        return false;
      else
      { System.err.println("conc()");
        synchronized (this)
        { got=CONC;
          notifyenv(); }
        list.cons(i);
        amount+=i.price();
```

```
      while (!cp.ack())
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      return true; }
  }

  public boolean get()
  {
    if ((GET&acc)!=GET)
      return false;
    else
    { System.err.println("get()");
      synchronized (this)
      { got=GET;
        notifyenv(); }
      list=null;
      amount=0;
      while (!rc.to())
      { synchronized (this)
        { try { wait(); }
          catch (InterruptedException ignored) { } } }
      return true; }
  }

  public void run()
  {
    System.err.println("runcur");
    while ((got&GET)!=GET)
      accept(CONC|GET);
  }
}

class DayTotal extends Foose
{
  private final static int ADD=1;
  private IList list;

  public DayTotal()
  {
    System.out.println("DayTotal");
  }

  public void reg(Object[] e)
  {
    env=e;
  }

  public boolean add(Item i)
  {
    if ((ADD&acc)!=ADD)
      return false;
    else
    { System.out.println("add()");
      synchronized (this)
      { got=ADD;
        notifyenv(); }
      list.cons(i);
```

```
      return true; }
  }

  public void run()
  {
    System.out.println("rundt");
    while (true)
      accept(ADD);
  }
}

class User extends Thread
{
  private CustomerPanel cp;
  private Receiver rc;
  private Current cur;
  private DayTotal dt;
  private Object[] env;

  public void run()
  {
    cp=new CustomerPanel();
    rc=new Receiver();
    cur=new Current();
    dt=new DayTotal();

    env=new Object[5];
    env[0]=cp;
    env[1]=rc;
    env[2]=cur;
    env[3]=dt;
    env[4]=this;

    cp.reg(rc,env);
    rc.reg(cp,cur,dt,env);
    cur.reg(cp,rc,env);
    dt.reg(env);

    dt.start();
    System.out.println("dt");
    cp.start();
    System.out.println("cp");
    rc.start();
    System.out.println("rc");
    cur.start();
    System.out.println("cur");

    while (!cp.mystart())
    { System.out.println("start?");
      synchronized (this)
      { try { wait(); }
        catch (InterruptedException ignored) { } } }
    while (!cp.myreturn(new CItem("Item 1")))
    { System.out.println("return1?");
      synchronized (this)
      { try { wait(); }
        catch (InterruptedException ignored) { } } }
    while (!cp.myreturn(new CItem("Item 2")))
    { System.out.println("return2?");
```

```
      synchronized (this)
      { try { wait(); }
        catch (InterruptedException ignored) { } } }
    while (!cp.myreturn(new CItem("Item 3")))
    { System.out.println("return3?");
      synchronized (this)
      { try { wait(); }
        catch (InterruptedException ignored) { } } }
    while (!cp.receipt())
    { System.out.println("receipt?");
      synchronized (this)
      { try { wait(); }
        catch (InterruptedException ignored) { } } }
  }
}

public class RecyclingMachine extends java.applet.Applet
{
  public void start()
  {
    System.out.println("Main");

    new Thread(new User()).start();
  }
}
```

The following is a sample debugging output of this Java program (on `System.err`):

```
Main           return()       ack()
CustomerPanel  new()          receipt?
Receiver       add()          receipt()
Current        conc()         printreceipt()
DayTotal       ack()          get()
dt             return2?       to()
cp             return()       send()
rc             new()
cur            add()
start?         conc()
rundt          ack()
runcp          return3?
runrc          return()
runcur         new()
start()        add()
return1?       conc()
```

Electronic Notes in Theoretical Computer Science 4 (1996)

An Actor Rewriting Theory

Carolyn L. Talcott

Computer Science Department, Stanford University
Stanford, CA 94305
http://www-formal.stanford.edu/clt/home.html

Abstract

We present a semantic framework for actor systems based on rewriting logic. This framework accounts for fairness and provides a variety of semantics for actor system components that have good composability properties.

1 Introduction

We are interested in developing a semantic foundation for open distributed systems that supports specifying, composing, and reasoning about components of open distributed systems. We take the actor model of computation [12,1,2] as our starting point. Actors are independent computational agents that interact solely via message passing. An actor can create other actors; send messages; and modify its own local state. An actor can only effect the local state of other actors by sending them messages, and it can only send messages to its acquaintances – addresses of actors it was given upon creation, it received in a message or actors it created. Actor semantics requires computations to be fair.

We take two views of actors: as individuals and as elements of components. Individual actors provide units of encapsulation and integrity. Components are collections of actors (and messages) provided with an interface specifying the receptionists, actors accessible from outside the component. Collecting actors into components provides for composability and coordination. Individual actors are described in terms of local transitions. Components are described in terms of interactions with their environment. A component may add an internal actor to its set of receptionists by including its address in an outgoing message.

To avoid making a specific choice of programming language to describe individual actor behaviors, we introduce the notion of abstract actor structure (AAS). An AAS provides an abstract set of states of individual actors and functions that determine the local transitions of individual actors. Using techniques of concurrent rewriting semantics [16] the semantics of components is derived from the local semantics of individual actors. This gives a concurrent computational system that is the basis for further semantic development.

© *1996 Elsevier Science B. V.*

For example, the interaction semantics of actor components [21] is defined by ignoring the details of internal transitions. Interaction semantics is a composable semantics with many pleasant properties. Also, the semantics of the higher-order actor language studied in [3] can be easily reformulated in the AAS/rewriting framework.

Abstract actor structures give an axiomatic characterization of actor languages, thus providing a basis for reasoning about heterogeneous systems. Concurrent rewriting semantics provides a truly concurrent semantics for actor systems. Interleaving semantics can be naturally embedded thus allowing us to work with either form depending on what is most convenient. The rewriting logic framework allows one to give a concise local presentation of the rules governing individual actors, and to derive the behavior of components from this. Equations abstracting states and computation paths allows one to treat actor computations at many levels of detail depending on the particular need.

Overview

In this paper we define an Actor rewrite theory, Rt_{A}, which is parametric in the choice of a particular abstract actor structure. The initial model construction of finite computations is extended with a notion of fair paths (infinite computations). This requires some work, since in Rt_{A} states the addresses of non-receptionist actors of a component are hidden in much the same way that names of bound variables are hidden in the lambda calculus. We associate to each path a set of *open forms* corresponding to a consistent choice of addresses for internal actors, and say that a path is *observationally fair* if it has an open form that is fair in the usual sense of fairness for actor computations. This is done by defining a flat actor rewrite theory, ${}^{\flat}Rt_{\mathrm{A}}$, with essentially the same rewrite rules, but without any hiding of addresses and extending the initial model with notions of admissible and fair paths. We then define a rewrite theory map from ${}^{\flat}Rt_{\mathrm{A}}$ to Rt_{A}. The set of open forms of an Rt_{A} computation is its inverse image under this map. These rewriting theories give rise to a variety of path-set semantics for actor system components. A path-set semantics associates to each component a set of computations having that component as source. Examples are finite computations, paths, fair paths, open or with hiding. As pointed out above, there is a natural notion of parallel composition of actor components. We further extend the rewriting semantics by defining a composition operation on path sets and show that the various path set semantics are compositional: for composable components the path set of their composition is the composition of their path sets. This is accomplished by defining composition for path sets of ${}^{\flat}Rt_{\mathrm{A}}$ and lifting the results to Rt_{A} using the rewrite theory map.

A more abstract semantics called *interaction semantics* is presented in [21]. This is derived from the fair path set semantics by omitting details of internal computations. This abstracts paths to sequences of interactions—messages coming into and going out of a component. Using the results of the present paper we can establish a compositionality theorem for interaction semantics.

Details will appear elsewhere.

The remainder of this paper is organized as follows. A brief summary of the notions of rewriting logic extended with notation for infinite computations is given in §2. Abstract actors structures are defined in §3. This is a minor variant of the definition given in [21]. the actor rewrite theory, Rt_{A} is presented in §4. The flattened version, ${}^{\flat}Rt_{\mathrm{A}}$, and the unflattening map to Rt_{A} is given in §5, along with a definition of fairness. Composition and compositionality are dealt with in §6. Directions for future research are discussed in §7.

We assume that the reader is familiar the basics of rewriting logic and its use to model concurrent computation (cf. [15,16,19]). We focus here on the development of the actor rewrite semantics. For examples of actor systems expressed and composed in this framework see [21]. We conclude this introduction with a brief summary of ourmathematical notation conventions. Let Y, Y_0, Y_1 be sets. We specify meta-variable conventions in the form: let y range over Y, which should be read as: the meta-variable y and decorated variants such as y', y_0, ..., range over the set Y. Y^n is the set of sequences of elements of Y of length n. Y^* is the set of finite sequences of elements of Y. $\bar{y} = [y_1, \ldots, y_n]$ is the sequence of length n with ith element y_i. (Thus $[\,]$ is the empty sequence.) $\mathbf{P}_\omega[Y]$ is the set of finite subsets of Y. $[Y_0 \stackrel{\circ}{\to} Y_1]$ is the set of partial functions, f, with domain, $\mathrm{Dom}(f)$, contained in Y_0 and range, $\mathrm{Rng}(f)$, contained in Y_1, $[Y_0 \to Y_1]$ is the set of $f \in [Y_0 \stackrel{\circ}{\to} Y_1]$ such that $\mathrm{Dom}(f) = Y_0$, and $\mathbf{Bij}(Y)$ is set of $f \in [Y \to Y]$ that are bijections. **Nat** is the set of natural numbers.

2 Rewriting Preliminaries

A rewriting theory, $Rt = (Eth, Rules)$, consists of an (order sorted) equational theory $Eth = (Sig, Eqns)$ over a given set of variables, Var, together with a labelled set of rewrite rules, $Rules$. Operations of our actor rewrite theory are partial in the sense that only applications to arguments meeting certain constraints are considered to be well-formed. To introduce an operation g such that $g(x, y)$ is well-formed and of sort Z just if x is of sort X, y is of sort Y, and the condition $\Phi(x, y)$ holds we write (assuming declarations $x : X$, $y : Y$)

$$g(_,_) : X \times Y \stackrel{\circ}{\to} Z$$
$$g(x, y) : Z \quad \text{if} \quad \Phi(x, y)$$

This notation is supported by the notion of *Membership Algebra* (cf. [18]) and is an informal version of the equational part of the Maude language [17]. If an operation has been introduced in this manner, then when we write $g(tm_x, tm_y)$ it is to be understood that the term is well-formed, i.e. that $\Phi(tm_x, tm_y)$ holds.

Let Tm be the equivalence classes of terms $(Sig, Eqns)$ with variables in Var. We write $tm(\vec{x})$ to specify an enumeration of the variables in tm and $tm(\overrightarrow{tm})$ for the substitution of the i-th element of $\overrightarrow{tm}$ for the i-th variable in $\vec{x}$. The initial model construction (see [16], §3. or [19], §3.) gives us a set,

$Cmp[Rt]$, of finite computations. Each computation c has a source, $src(c)$, and a target, $tgt(c)$, in Tm and we write $c : tm_0 \Rightarrow tm_1$ to indicate that $src(c) = tm_0$ and $tgt(c) = tm_1$. An instantiation of a rule, $r(\vec{x}) : tm_0(\vec{x}) \Rightarrow tm_1(\vec{x})$, has the form $r(\overrightarrow{c}) : tm_0(\overrightarrow{tm}) \Rightarrow tm_1(\overrightarrow{tm'})$ where $\overrightarrow{c} : \overrightarrow{tm} \Rightarrow \overrightarrow{tm'}$ is a list of computations (of appropriate sorts and length). Each rule instantiation is in $Cmp[Rt]$ and $Cmp[Rt]$ is closed under the operations of Sig. We assume that sort constraints of the equational part of Rt are lifted to computation terms, so that a computation term is considered well-formed just if both the source and target terms are well-formed. For each $tm \in Tm$ the identity computation $\mathtt{id}[tm] : tm \Rightarrow tm$ is in $Cmp[Rt]$. We adopt the convention that when tm appears in a context requiring a computation it stands for $\mathtt{id}[tm]$. If $c_0 : tm_0 \Rightarrow tm_1$, and $c_1 : tm_1 \Rightarrow tm_2$ are computations, then $c_0 \,;\, c_1 : tm_0 \Rightarrow tm_2$ is a computation (closure under sequential composition). The initial model construction also gives us an equivalence relation, $\sim$, on computations. $\sim$ satisfies $Eqns$, laws saying that computations are the arrows of a category, functoriality laws for the operations of Sig, and exchange laws that allow concurrent computations to be put into sequential form.

Since an important aspect of the actor model of computation is fairness, we need to consider infinite as well as finite computations. We call infinite computations *paths*. Given a set of (finite) computations Q, The paths over Q, $Path[Q]$, is the set of infinite sequences from Q such that adjacent computations are sequentially composable. The source term of a path, $src(p)$, is the source of its initial finite computation. The initial segment up to i of a path, $p\lceil i$ is the sequential composition of the path elements up to and including $p(i)$.

$$Path[Q] = \{p \in \mathbf{Nat} \to Q \mid (\forall i \in \mathbf{Nat}) tgt(p(i)) = src(p(i+1))\}$$
$$src(p) = src(p(0))$$
$$p\lceil i = p(0) \,;\, \ldots \,;\, p(i)$$

For any set, S, of (finite or infinite) computations we define the S computations from tm, $S(tm)$ by

$$S(tm) = \{s \in S \mid src(s) = tm\}$$

We extend this definition to sets of terms, T, in the usual way: $S(T) = \{s \in S \mid (existstm \in T) src(s) = tm\}$. The equivalence $\sim$ on finite computations is lifted to paths as follows.

Definition (Path equivalence): For $p, p' \in Path[Q]$

$$p \sim p' \Leftrightarrow (\forall i)(\exists k, c) p\lceil i \,;\, c \sim p'\lceil (i+k) \;\wedge\; (\forall i)(\exists k, c) p'\lceil i \,;\, c \sim p\lceil (i+k)$$

If $\mathtt{id}[tm] \,;\, c \sim c$, then inserting finitely many (possibly zero) identity computations before each computation in a path produces an equivalent path. Also if $p(i) \sim p'(i)$ for $i \in \mathbf{Nat}$ then clearly $p \sim p'$. This treatment of infinite computations over a rewrite theory is similar to the treatment of infinite computations in the special case of Petri nets given in [20].

3 Abstract Actor Structures

An actor system component consists of a collection of actors interacting with each other and with their enviornment via asynchronous message passing. Thus we can think of a component as a multiset of actors and messages encapsulated to ensure that only certain specified actors, the receptionists, receive messages from outside the component. The behavior of a component is determined by the behavior of the individual actors. The actor rewrite theory given in §4 is one means of describing this collected behavior. To describe individual actor behavior we abstract from the details of a particular programming language and introduce the notion of abstract actor structure (AAS). The gives an axiomatic characterization of the admissible semantics of an actor language.

An abstract actor structure (AAS) is a structure with sorts, relations, and operations as follows.

Sorts: $\mathbf{A}$, $\mathbf{V}$, $\mathbf{S}$, and $\mathbf{oF}$, with $\mathbf{A} \subseteq \mathbf{V}$.
Relations: $En_{\mathrm{d}} \subseteq \mathbf{A} \times \mathbf{S} \times \mathbf{V}$, $En_{\mathrm{ex}} \subseteq \mathbf{A} \times \mathbf{S}$;
Operations:

$(_)_ : \mathbf{A} \times \mathbf{S} \to \mathbf{oF}$
$_ \lhd _ : \mathbf{A} \times \mathbf{V} \to \mathbf{oF}$
$\diamond : \mathbf{oF}$
$\{_, _\} : \mathbf{oF} \times \mathbf{oF} \stackrel{\circ}{\to} \mathbf{oF}$
$recep : \mathbf{oF} \to \mathbf{P}_\omega[\mathbf{A}]$
$acq : \mathbf{S} \cup \mathbf{V} \to \mathbf{P}_\omega[\mathbf{A}]$
$Deliv : En_{\mathrm{d}} \to \mathbf{oF}$
$Ex : En_{\mathrm{ex}} \to \mathbf{A}^* \stackrel{\circ}{\to} \mathbf{oF}$,
$\#new : En_{\mathrm{ex}} \to \mathbf{Nat}$
$\widehat{(\,)} : \mathbf{Bij}(\mathbf{A}) \to [\mathbf{S} \cup \mathbf{V} \to \mathbf{S} \cup \mathbf{V}]$

$\mathbf{A}$ is a countable set of actor addresses, $\mathbf{V}$ is a set of values that can be communicated between actors, and $\mathbf{S}$ is a set of actor states. Actor addresses are among the values that can be communicated. Actor states are intended to carry information traditionally contained in the script (methods) and acquaintances (instance variables), as well as the local message queue and the current processing state. $\mathbf{oF}$ is the set of open fragments—multisets of actors and messages in which no two actor occurrences have the same address. Open fragments model the internal structure of an actor system. We let a range over $\mathbf{A}$, v range over $\mathbf{V}$, s range over $\mathbf{S}$, and oF range over $\mathbf{oF}$.

$(s)_a$ is an *actor* with address, a, in state, s. $a \lhd v$ is a *message* with target, a, and contents, v. $recep(oF)$ is the set of addresses of actors occurring in oF. (We call these the receptionists since in an open fragment all actors are visible to the environment.) $\{oF_0, oF_1\}$ is the multiset union of oF_0 and oF_1 provided that the receptionists of oF_0 and oF_1 are disjoint.

$En_{\mathrm{d}}(a, s, v)$ is a predicate on actors and values that holds if the actor, $(s)_a$, is enabled for delivery of a message with contents v. If $En_{\mathrm{d}}(a, s, v)$, then $Deliv(a, s, v)$ is the actor with address a in a state resulting from delivery of the message.

We need to be able to determine the actor addresses known to an actor or communicated in a message. Since states and values are abstract entities, a means of determining the actor addresses occurring in states and values is needed. This is met by the acquaintance function, acq, which gives the (finite) set of actor addresses occurring in a state or value.

$En_{\mathrm{ex}}(a, s)$ is a predicate on actors that holds if actor a in state s is enabled for execution, and $Ex((s)_a)$ specifies the result of a single execution step by actor $(s)_a$. Since new actors may be created and the addresses of these new actors can only be determined at 'runtime' we formulate the local execution semantics, $Ex((s)_a)$, as a function from lists of actor addresses to open fragments. $\#new(s)$ is the number of new actors that will be created by actor $(s)_a$ executing a step and If $En_{\mathrm{ex}}(a, s)$ and $\vec{a} = [a_1, \ldots a_{\#new(s)}]$ is a list of actor addresses distinct from one another and from the a and $acq(s)$, then $Ex((s)_a)[a_1, \ldots a_{\#new(s)}]$ is the fragment produced. This fragment must contain the actor with address a, possibly with a modified state, and an actor with address a_i for $1 \leq i \leq \#new(s)$. It may also contain messages to new actors or to acquaintances of $(s)_a$.

As a simple example, we can model a forever idling actor by a state `idle` with no acquaintances, enabled for execution but not delivery, such that $\#new(\mathtt{idle}) = 0$ and $Ex((\mathtt{idle})_a)[\,] = (\mathtt{idle})_a$. As a slightly less trivial example let us model the behavior of a factory actor that accepts requests for new actors with some fixed behavior, creates such an actor and send a message containing the address of the newly created actor to the requestor. Let B be the initial state of the desired behavior. The factory actor behavior is describe by a state `fac` and a family of states $\mathtt{facx}(a')$ for $a' \in \mathbf{A}$ such that $(\mathtt{fac})_a$ is enabled only for delivery of messages whose contents is an actor address. $(\mathtt{facx}(a'))_a$ is enabled only for execution, with $\#new(a, \mathtt{facx}(a')) = 1$, $acq(\mathtt{facx}(a')) = \{a'\}$, $Deliv(a, \mathtt{fac}, a') = (\mathtt{facx}(a'))_a$, and

$$Ex((\mathtt{facx}(a'))_a)[a_0] = \{(\mathtt{fac})_a\,,\ (B)_{a_0}\,,\ a' \lhd a_0\}.$$

Actor addresses cannot be explicitly created by actors, and the semantics cannot depend on the particular choice of addresses of a group of actors. A renaming mechanism is used to formulate this requirement. We let ρ range over bijections on $\mathbf{A}$ (renamings). For any such ρ, $\widehat{\rho}$ is a renaming function on states and values that agrees with ρ on actor addresses. Renaming is extended homomorphically to structures built from addresses, states, and values.

An AAS must obey the fundamental acquaintance laws of actors [4,6]. In particular, the axioms (**AR**) and (**Ex**) below must hold in an AAS. We begin with axioms specifying well-formed open fragments and the receptionist and renaming operations on open fragments, (**OF**).

Open Fragment Axioms (OF). $\{_\,,\ _\}$ is associative and commutative

with identity: $\diamond$.

$\{oF_0, oF_1\} : \mathbf{oF}$ if $recep(oF_0) \cap recep(oF_1) = \emptyset$
$recep((s)_a) = \{a\}$
$recep(a \lhd v) = \emptyset$
$recep(\diamond) = \emptyset$
$recep(\{oF_0, oF_1\}) = recep(oF_0) \cup recep(oF_1)$
$\widehat{\rho}(a \lhd v) = \rho(a) \lhd \widehat{\rho}(v)$
$\widehat{\rho}((s)_a) = (\widehat{\rho}(s))_{\rho(a)}$
$\widehat{\rho}(\diamond) = \diamond$
$\widehat{\rho}(\{oF_0, oF_1\}) = \{\widehat{\rho}(oF_0), \widehat{\rho}(oF_1)\}$

Acquaintance and Renaming Axioms (AR).

(_) $\widehat{\rho}(a) = \rho(a) \wedge \widehat{\rho}(s) \in \mathbf{S} \wedge \widehat{\rho}(v) \in \mathbf{V}$
$acq(a) = \{a\}$
(i) $acq(\widehat{\rho}(x)) = \{\rho(a) \mid a \in acq(x)\}$ for $x \in \mathbf{S} \cup \mathbf{V}$
(ii) $acq(Deliv(a, s, v)) \subseteq \{a\} \cup acq(v) \cup acq(s)$ if $En_{\mathrm{d}}(a, s, v)$
(iii) $\widehat{\rho}(Deliv(a, s, v)) = Deliv(\rho(a), \widehat{\rho}(s), \widehat{\rho}(v))$ if $En_{\mathrm{d}}(a, s, v)$
(iv) $En_{\mathrm{ex}}(a, s) \Leftrightarrow En_{\mathrm{ex}}(\rho(a), \widehat{\rho}(s))$ and $En_{\mathrm{d}}(a, s, v) \Leftrightarrow En_{\mathrm{d}}(\rho(a), \widehat{\rho}(s), \widehat{\rho}(v))$
(v) $\#new(a, s) = \#new(\rho(a), \widehat{\rho}(s))$
(vi) $(\forall a \in acq(x))(\rho(a) = a) \Rightarrow \widehat{\rho}(x) = x$ for $x \in \mathbf{S} \cup \mathbf{V}$
(vii) $\widehat{\rho_0 \circ \rho_1} = \widehat{\rho_0} \circ \widehat{\rho_1}$

(i) and (iii) say that renaming commutes with the delivery and acquaintance functions. (ii) says that acquaintances of an actor in state, s, after delivery of a message with contents, v, are among the acquaintances of s and of v. (iv) and (v) say that renaming does not change enabledness or the number of actors that will be created upon execution. (vi) says that if a renaming fixes the acquaintances of an object then applying it does not change the object. (vii) says that the renaming mechanism commutes with function composition. (vi) and (vii) imply that two renamings that agree on the acquaintances of an object have the same result when applied to the object, and that $\widehat{\rho}$ is a bijection on $\mathbf{S} \cup \mathbf{V}$. Also, (_) says that $\widehat{\rho}$ fixes $\mathbf{A}$, $\mathbf{V}$, and $\mathbf{S}$.

Since the set of actor addresses occurring in any state or value is finite, we could have formulated the renaming requirements using finite maps , thus making an AAS truly a first-order structure. We have used infinite bijections in this presentation, since various constraints become a bit simpler.

Execution axioms (Ex). If $En_{\mathrm{ex}}(a, s)$ and if $\vec{a}$ is a list of $\#new(s)$ distinct actor addresses disjoint from a and $acq(s)$, then $Ex((s)_a)(\vec{a}) = oF$ for some $oF \in \mathbf{oF}$, such that

(i) $recep(oF) = \{a\} \cup \vec{a}$ the set of addresses of actors occurring in oF

(ii) $(s')_{a'} \in oF \Rightarrow acq(s') \subseteq acq(s) \cup recep(oF)$

(iii) $a' \lhd v' \in oF \Rightarrow \{a'\} \cup acq(v') \subseteq acq(s) \cup recep(oF)$

(iv) $Ex(\widehat{\rho}((s)_a))(\widehat{\rho}(\vec{a})) = \widehat{\rho}(oF)$

(ii) says that any acquaintance of an actor after executing a step or of a newly created actor either was an acquaintance of the actor before the step is taken, or is one of the newly created actors. (iii) says that the targets and contents of newly sent messages are similarly constrained. (iv) says that executing a step commutes with renaming – that is, the local semantics is uniformly parameterized by the set of locally occurring actor addresses.

4 The Actor Rewriting Theory, Rt_{A}

We assume given a fixed but unspecified abstract actor structure specification and define the actor rewrite theory, Rt_{A}, relative to this specification. It is easy to see that Rt_{A} is parametric in the choice of an AAS theory.

Rt_{A} has two layers: one for internal computation and one for interaction of an actor system with its environment. This separation reflects the different composability properties of internal computations and interactions. These differences arise because composition of interactions may internalize some interactions, and because composition of interactions is not allowed to lose messages output by one subcomponent to a receptionist of the other subcomponent.

4.1 The Equational part of Rt_{A}

The equational part, Eth_{A}, of Rt_{A}, extends the AAS equational theory. There are two additional sorts, fragments, $F \in \mathbf{F}$, and components, $C \in \mathbf{C}$ with $\mathbf{oF}$ a subsort of $\mathbf{F}$. Internal computation rules act on fragments and interaction rules act on components. Fragments are formed from messages and actors by multiset union, with the additional operation of receptionist restriction. Multiset union on fragments is also restricted to arguments that have disjoint receptionists. The receptionist operation, *recep*, and renaming application, $\widehat{\rho}$, are extended to fragments. There is an additional operation, $extrn(F)$—the addresses of actors mentioned in F but not defined there. Components are just fragments wrapped to isolate them from the fragment algebra.

Operations of Rt_{A}

$\{_\,,\,_\} : \mathbf{F} \times \mathbf{F} \stackrel{\circ}{\to} \mathbf{F}$ associative, commutative, with identity: $\diamond$, and

$\{F_0\,,\,F_1\} : \mathbf{F}$ if $recep(F_0) \cap recep(F_1) = \emptyset$

$_\lceil_ : \mathbf{F} \times \mathbf{P}_\omega[\mathbf{A}] \stackrel{\circ}{\to} \mathbf{F}$ and $F\lceil R : \mathbf{F}$ if $R \subseteq recep(F)$

$\langle_\rangle : \mathbf{F} \to \mathbf{C}$

$recep(_) : \mathbf{F} \to \mathbf{P}_\omega[\mathbf{A}]$

$extrn(_) : \mathbf{F} \to \mathbf{P}_\omega[\mathbf{A}]$

$\widehat{(\,)} : \mathbf{Bij}(\mathbf{A}) \to [\mathbf{F} \to \mathbf{F}]$

The equations of Eth_{A} extend the equations of the AAS and those implicit in the declaration that multiset union is associative and commutative with identity the empty fragment, with the axioms (**rcp.ren**) extending the receptionist and renaming application functions, (**ext**) axiomatizing the external address function, and (**rr**) axiomatizing restriction and renaming equality on fragments.

Receptionist axioms (rcp)

$$recep(\{F_0\,,\ F_1\}) = recep(F_0) \cup recep(F_1)$$
$$recep(F\lceil R) = R$$
$$\widehat{\rho}(\{F_0\,,\ F_1\}) = \{\widehat{\rho}(F_0)\,,\ \widehat{\rho}(F_1)\}$$
$$\widehat{\rho}(F\lceil R) = \widehat{\rho}(F)\lceil\widehat{\rho}(R)$$

External addresses axioms (ext)

$$extrn(\diamond) = \emptyset$$
$$extrn(a \lhd v) = acq(v) \cup \{a\}$$
$$extrn((s)_a) = acq(s) - \{a\}$$
$$extrn(\{F_0\,,\ F_1\}) = (extrn(F_0) \cup extrn(F_1)) - recep(F_0, F_1)$$
$$extrn(F\lceil R) = extrn(F)$$

Restriction and renaming axioms (rr)

(top) $\quad F = F\lceil recep(F)$

(erase) $\quad \{F_0\lceil R_0\,,\ F_1\}\lceil R = \{F_0\,,\ F_1\}\lceil R$
$\qquad\qquad$ if $(recep(F_0) - R_0) \cap extrn(F_1) = \emptyset$

(alpha) $\quad F\lceil R = \widehat{\rho}(F)\lceil R \qquad$ if ρ is the identity on R and $extrn(F)$

(**top**) says that there is an implicit restriction to the receptionists of a fragment. (**erase**) says that inner restrictions can be erased (or redrawn) since they do not change what is seen from the outside if it is protected by an explicit restriction. (**alpha**) is a form of alpha equivalence. It says that the interface wrapping operation, $F\lceil R$ hides the choice of names (addresses) internal actors not explicitly exported. Put another way, official choice of name can be postponed until such time as the actor address is exported.

Note that equality in **F** restricted to **oF** is just multiset equality. Also if $oF\lceil R = oF'\lceil R'$, then $R = R'$ and there is some renaming, ρ, that fixes R and $extrn(oF)$ and such that $oF' = \widehat{\rho}(oF)$. Every component has the form $\langle\, F \,\rangle$ for some fragment, F, and two components $\langle\, F \,\rangle$ and $\langle\, F' \,\rangle$ are equal just if $F = F'$.

Lemma (Open form): For any fragment F we can find oF and R such that $F = oF\lceil R$. Note that $R = recep(F)$.

Proof : By induction on the construction of F. $\diamond$, $(s)_a$, $a \lhd v$ are open fragments and can be put in open form using (**top**). If $F = F_0\lceil R$ let $oF_0\lceil R_0$ be an open form for F_0. Then by (**erase**) (and the multiset rules to introduce

and eliminate $\diamond$) $oF_0\lceil R$ is an open form for F. If $F = \{F_0\,,\ F_1\}$, then let $oF_j\lceil R_j$ be open forms of F_j for $j < 2$ such that $recep(oF_0) \cap recep(oF_1) = \emptyset$. Then $\{oF_0\,,\ oF_1\}\lceil R_0 \cup R_1$ is an open form of F. □

4.2 Rules of Rt_{A}

Internal actor computation rules. The internal rules of Rt_{A} are the execution rule, (**exe**), and the delivery rule, (**del**).

(exe) $\mathtt{e}(a,s) : (s)_a \Rightarrow Ex((s)_a)(\vec{a})\lceil\{a\}$ if $En_{\mathrm{ex}}(a,s)$ and $\vec{a}$ is a list of $\#new(a,s)$ distinct actor addresses disjoint from $a, acq(s)$

(del) $\mathtt{d}(a,s,v) : \{(s)_a\,,\ a \lhd v\} \Rightarrow Deliv(a,s,v)$ if $En_{\mathrm{d}}(a,s,v)$

Component interaction rules. There are two interaction rules for Rt_{A}: (**in**) for incoming messages; and (**out**) for outgoing messages.

(in) $\mathtt{i}(F, a \lhd v) : \langle\, F \,\rangle \Rightarrow \langle\, \{F\,,\ a \lhd v\} \,\rangle$ if $a \in recep(F)$

(out) $\mathtt{o}(F, R, a \lhd v) : \langle\, \{F\,,\ a \lhd v\}\lceil R \,\rangle \Rightarrow \langle\, F\lceil R \cup (acq(v) \cap recep(F)) \,\rangle$ if $a \notin recep(F)$

Definition (The Finite Actor Rewrite Theory): Rt_{A} is the rewrite theory with equational part Eth_{A} and rules (**exe,del,in,out**).

We let $Cmp_{\mathrm{A}} = Cmp[Rt_{\mathrm{A}}]$. Using the definition given in §2, $Cmp_{\mathrm{A}}(\mathbf{F})$ is the set of computations with source (and target) in $\mathbf{F}$. We call these internal computations and we let τ range over $Cmp_{\mathrm{A}}(\mathbf{F})$. Similarly, $Cmp_{\mathrm{A}}(\mathbf{C})$ is the set of computations acting on components and we let γ range over $Cmp_{\mathrm{A}}(\mathbf{C})$. Applying the inital model construction to Rt_{A} we see that Cmp_{A} is generated from the internal rules (**exe**) and (**del**) (whose only instances are with identity computations for parameters) by the following clauses:

(id) $\mathtt{id}[X] : X \Rightarrow X$

(scmps) $\xi_0\,;\xi_1 : X_0 \Rightarrow X_2$ if $\xi_0 : X_0 \Rightarrow X_1$ and $\xi_1 : X_1 \Rightarrow X_2$

(mun) $\{\tau_0\,,\ \tau_1\} : \{F_0\,,\ F_1\} \Rightarrow \{F_0'\,,\ F_1'\}$ if $recep(F_0) \cap recep(F_1) = \emptyset$ and $\bigwedge_{j<2} \tau_j : F_j \Rightarrow F_j'$

(rf) $\tau\lceil R : F\lceil R \Rightarrow F'\lceil R$ if $R \subseteq recep(F)$ and $\bigwedge_{j<2} \tau_j : F_j \Rightarrow F_j'$

(in) $\mathtt{i}(\tau, a \lhd v) : \langle\, F \,\rangle \Rightarrow \langle\, \{F'\,,\ a \lhd v\} \,\rangle$ if $\tau : F \Rightarrow F', a \in recep(F)$

(out) $\mathtt{o}(\tau, R, a \lhd v) : \langle\, \{F\,,\ a \lhd v\}\lceil R \,\rangle \Rightarrow \langle\, F'\lceil R \cup (acq(v) \cap recep(F)) \,\rangle$ if $\tau : F \Rightarrow F', a \notin recep(F)$

where X, X_j range over $\mathbf{F}$ or $\mathbf{C}$ and ξ, ξ_j range over computations on fragments or computations as appropriate.

Computations on fragments preserve receptionists and may decrease externals. Computations on components are non-decreasing on receptionists.

Lemma (recep.extern.cmp):
(1) If $\tau : F \Rightarrow F'$, then $recep(F) = recep(F')$ and $extrn(F) \supseteq extrn(F')$.
(1) If $\gamma : \langle\, frag \,\rangle \Rightarrow \langle\, F' \,\rangle$, then $recep(F) \subseteq recep(F')$.

The equivalence relation $\sim$ on computations is a congruence that satisfies the axioms (**cat**), (**funct**), (**epart**), and (**exch**). (**cat**) states that $_\ ;\ _$ is associative with left and right identity the appropriate identity computations. (**funct**) states that the fragment and component forming operations are functorial — preserve identities and sequential composition. For example, functoriality of $\{_,\ _\}$ means $\{\mathtt{id}[F],\ \mathtt{id}[F']\} \sim \mathtt{id}[\{F,\ F'\}]$ and $\{\tau_1 ; \tau_2,\ \tau_3 ; \tau_4\} \sim \{\tau_1,\ \tau_3\}\ ;\ \{\tau_2,\ \tau_4\}$ (when both sides are well-formed). (**epart**) states that the equational axioms for fragments lift to computations. In addition to multi-set axioms for $\{_,\ _\}$ there are the axioms for restriction and renaming. The lifting of the axioms (**rr**) gives

(top') $\quad \tau\lceil src(\tau) \sim \tau$
(erase') $\quad \{\tau_0\lceil R_0,\ \tau_1\}\lceil R \sim \{\tau_0,\ \tau_1\}\lceil R \qquad$ if $(recep(src(\tau_0)) - R_0) \cap extrn(src(\tau_1)) = \emptyset$
(alpha') $\quad \widehat{\rho}(\tau)\lceil R \sim \tau\lceil R \qquad$ if ρ is the identity on R and $extrn(src(\tau))$

Finally there are the exchange axioms that allow the actions of a computation to be sequentialized.
Exchange (exch): If $\tau : F \Rightarrow F'$, then

(1) $\mathtt{i}(\tau, a \lhd v) \sim \langle\, \tau \,\rangle\ ;\ \mathtt{i}(\mathtt{id}[F'], a \lhd v) \sim \mathtt{i}(\mathtt{id}[F], a \lhd v)\ ;\ \langle\, \{\tau,\ \mathtt{id}[a \lhd v]\} \,\rangle$
(2) $\mathtt{o}(\tau, R, a \lhd v) \sim \langle\, \{\tau,\ a \lhd v\}\lceil R \,\rangle\ ;\ \mathtt{o}(\mathtt{id}[F'], a \lhd v)$
$\qquad \sim \mathtt{o}(\mathtt{id}[F], R, a \lhd v)\ ;\ \langle\, \tau\lceil R \cup (acq(v) \cap recep(F)) \,\rangle$

4.3 Infinite computations

The admissible paths of Rt_{A} are paths that satisfy a global addressing constraint ensuring that in an admissible path no forgotten external address can be reused as a receptionist address, and that the address space is not used up. To make this precise we define, for any (finite or infinite) computation c, the (global) externals, $extrn(c)$, and receptionists, $recep(c)$ by

$$extrn(\gamma) = extrn(src(\gamma)) \qquad \text{if } \gamma \text{ is constructed without } _\ ;\ _$$
$$extrn(\gamma_0\ ;\ \gamma_1) = extrn(\gamma_0) \cup extrn(\gamma_1)$$
$$extrn(p) = \bigcup_{i \in \mathbf{Nat}} extrn(p(i))$$
$$recep(\gamma) = recep(src(\gamma))$$
$$recep(\gamma_0\ ;\ \gamma_1) = recep(\gamma_0) \cup recep(\gamma_1)$$
$$recep(p) = \bigcup_{i \in \mathbf{Nat}} recep(p(i))$$

The admissible paths are then defined by

$$Gac(p) \Leftrightarrow extrn(p) \cap recep(p) = \emptyset \wedge Countable(\mathbf{A} - (extrn(p) \cup recep(p)))$$

$$Path_{\mathrm{A}} = \{p \in Path[Cmp_{\mathrm{A}}] \mid Gac(p)\}$$

We let p range over $Path_{\mathrm{A}}$. Note that $Path_{\mathrm{A}}$ is closed under $\sim$ since equivalence preserves receptionists and externals.

To complete the development of the actor rewrite theory, we need to say what the fair paths are. An event in an actor computation is an actor execution step or message delivery (including delivery to the environment). In a fair actor computation an event that is enabled at some point must eventually occur or be permanently disabled. In Rt_{A} we have carefully defined components to satisfy the requirement that non-receptionist actors cannot be referred to externally, since multiset equations prevent reference by position, and the renaming equations prevent reference by address. Thus we cannot directly say when an internal event is enabled in a component or when it fires in a computation. To circumvent this dilemma we introduce a notion of *observational fairness*—fairness relative to some consistent choice of addresses for internal actors. The assignment of consistent choices of addresses is accomplished by defining a less abstract rewriting semantics, ${}^{\flat}Rt_{\mathrm{A}}$ (the flat actor rewrite theory), in which there is no renaming equivalence. In this theory it is straight forward to define fairness. ${}^{\flat}Rt_{\mathrm{A}}$ is mapped to Rt_{A} by surjective rewrite theory map. The inverse image of a path under this mapping can be thought of as the collection of consistent choices for internal actor addresses. The observationally fair paths of Rt_{A} are the images of fair paths in ${}^{\flat}Rt_{\mathrm{A}}$.

5 The Flat Actor Rewrite Theory

The flat actor rewrite theory, ${}^{\flat}Rt_{\mathrm{A}}$, is defined by recasting the computation rules to act on flat components—components formed from open fragments and receptionist sets, equated only by multiset equations. Thus addresses of all actors defined in a component are fixed at creation time. The resulting computations correspond closely to the labeled transition system semantics for an actor actor language given in [3]. Fairness of actor computations in this setting is defined in the usual way (cf. [3]). ${}^{\flat}Rt_{\mathrm{A}}$ is mapped to Rt_{A} essentially by adding the renaming equations (**rr**), and mapping the recast rules to their originations. By initiality, this unflattening map lifts to computations (cf. [15]). Every component and computation (finite or infinite) is the image of a flat one (modulo equivalence), and we say that a path is observationally fair if it is the image of a fair path in ${}^{\flat}Rt_{\mathrm{A}}$.

5.1 The Equational Part

The equational part, ${}^{\flat}Eth_{\mathrm{A}}$ of ${}^{\flat}Rt_{\mathrm{A}}$, extends the AAS equational theory with two sorts, ${}^{\flat}\mathbf{F}$ and ${}^{\flat}\mathbf{C}$, and operations $_ \,/\, _$ to construct flat fragments from open fragments and receptionists, and $\langle _ \rangle$ to construct flat components from flat fragments.

$$_ \,/\, _ : \mathbf{oF} \times \mathbf{P}_{\omega}[\mathbf{A}] \stackrel{\circ}{\to} {}^{\flat}\mathbf{F}$$
$$oF \,/\, R : {}^{\flat}\mathbf{F} \quad \text{if} \quad R \subset recep(oF)$$

$\langle _ \rangle : {}^{\flat}\mathbf{F} \to {}^{\flat}\mathbf{C}$

We let ${}^{\flat}F$ range over ${}^{\flat}\mathbf{F}$ and ${}^{\flat}C$ range over ${}^{\flat}\mathbf{C}$.

5.2 Rules and finite computations

The internal computation rules act on open fragments while the interaction rules act on flat components.

(fexe) $\mathtt{e}(a, s, \vec{a}) : (s)_a \Rightarrow Ex((s)_a)(\vec{a})$
if $En_{\mathrm{ex}}(a, s)$ and $\vec{a}$ is a list of $\#new(s)$ distinct addresses disjoint from $a, acq(s)$

(fdel) $\mathtt{d}(a, s, v) : \{(s)_a\,,\ a \lhd v\} \Rightarrow Deliv(a, s, v)$ if $En_{\mathrm{d}}(a, s, v)$

(fin) $\mathtt{fi}(oF, R, a \lhd v) : \langle\, oF \,/\, R \,\rangle \Rightarrow \langle\, \{oF\,,\ a \lhd v\} \,/\, R \,\rangle$
if $a \in R$ and $acq(v) \cap recep(oF) \subseteq R$

(fout) $\mathtt{fo}(oF, R, a \lhd v) : \langle\, \{oF\,,\ a \lhd v\} \,/\, R \,\rangle \Rightarrow \langle\, oF \,/\, R \cup E \,\rangle$
if $a \notin recep(oF)$ and $E = acq(v) \cap recep(oF)$

The flat computations $Cmp[{}^{\flat}Rt_{\mathrm{A}}]$ are generated from the internal rules in a manner similar to the generation of $Cmp[Rt_{\mathrm{A}}]$. To be a faithful model of actor computation, we need to restrict application of the multiset union operation on computations, and the instantiation of the interaction rules. Specifically, we need to avoid addressing conflicts that might arise if an open fragment computation generates new actors and gives them addresses already occurring as external actor in fragment or message with which it is to be combined. This restriction gives the subclass of admissible finite computations, ${}^{\flat}Cmp_{\mathrm{A}} \subseteq Cmp[{}^{\flat}Rt_{\mathrm{A}}]$. We let δ range over ${}^{\flat}Cmp_{\mathrm{A}}(\mathbf{oF})$, ${}^{\flat}\tau$ range over ${}^{\flat}Cmp_{\mathrm{A}}({}^{\flat}\mathbf{F})$, and ${}^{\flat}\gamma$ range over ${}^{\flat}Cmp_{\mathrm{A}}({}^{\flat}\mathbf{C})$. (Recall our convention that ${}^{\flat}Cmp_{\mathrm{A}}(\mathbf{oF})$ is the set of admissible computations with source in $\mathbf{oF}$, etc.)

(omun) $\{\delta_0\,,\ \delta_1\} : \{oF_0\,,\ oF_1\} \Rightarrow \{oF'_0\,,\ oF'_1\}$
if $\delta_j : oF_j \Rightarrow oF'_j$
and $\bigwedge_{j<2}(extrn(oF'_j) \cap recep(oF'_{1-j}) \subset recep(oF_{1-j}))$

(in) $\mathtt{fi}(\delta, R, a \lhd v) : \langle\, oF \,/\, R \,\rangle \Rightarrow \langle\, \{oF'\,,\ a \lhd v\} \,/\, R \,\rangle$
if $\delta : oF \Rightarrow oF'$, $a \in R$, and $acq(v) \cap recep(oF') \subseteq R$

(out) $\mathtt{fo}(\delta, R, a \lhd v) : \langle\, \{oF\,,\ a \lhd v\} \,/\, R \,\rangle \Rightarrow \langle\, oF' \,/\, R \cup (acq(v) \cap recep(oF)) \,\rangle$
if $\delta : oF \Rightarrow oF'$, $a \notin recep(oF')$
and $acq(v) \cap recep(oF') \subseteq recep(oF)$

Note that ${}^{\flat}Cmp_{\mathrm{A}}$ is closed under the equivalence relation, $\sim$, on $Cmp[{}^{\flat}Rt_{\mathrm{A}}]$, since equivalent computations have the same source and target (classes). Also, each ${}^{\flat}\tau \in {}^{\flat}Cmp_{\mathrm{A}}({}^{\flat}\mathbf{F})$ has the form $\delta \,/\, R$.

5.3 Infinite Computations and Fairness

The infinite computations for flat components, ${}^{\flat}Path_{\mathrm{A}}$, are paths over ${}^{\flat}Cmp_{\mathrm{A}}$ that obey the global address constraint, where *extrn*, *recep*, and *Gac* are defined on $Path[{}^{\flat}Cmp_{\mathrm{A}}]$ in just the same way as they were defined on $Path[Cmp_{\mathrm{A}}]$.

Definition (Paths):

$$ {}^{\flat}Path_{\mathrm{A}} = \{{}^{\flat}p \in Path[{}^{\flat}Cmp_{\mathrm{A}}] \mid Gac({}^{\flat}p)\} $$

To define fairness we first define what it means for an actor or message to be *enabled* in a flat component, and what it means for an enabled actor or message to *fire* in a finite computation. Since we cannot force the environment to produce messages, inputs are ignored for the purpose of defining fairness.

We say that $a \in recep(oF)$ is enabled, written $Enabled(oF, (s)_a)$, if the actor, $(s)_a$, in oF is enabled for execution. Similarly $a \triangleleft v$ in oF is enabled, written $Enabled(oF, a \triangleleft v)$, if a is external, or if a is internal and the actor with address a in oF is enabled for delivery of a message with contents v.

Definition (*Enabled*):

$$ Enabled(oF, (s)_a) \Leftrightarrow (\exists oF_0)(oF = \{oF_0\,,\ (s)_a\} \wedge En_{\mathrm{ex}}(a, s)) $$

$$ \begin{aligned} &Enabled(oF, a \triangleleft v) \\ &\quad \Leftrightarrow (\exists oF_0)(oF = \{oF_0\,,\ a \triangleleft v\} \wedge \\ &\qquad\qquad (a \in extrn(oF) \vee \\ &\qquad\qquad\ (\exists oF_1, s)(oF_0 = \{oF_1\,,\ (s)_a\} \wedge En_{\mathrm{d}}(a, s, v)))) \end{aligned} $$

$$ Enabled(\langle\, oF \,/\, R \,\rangle, x) \Leftrightarrow Enabled(oF, x) $$

Let ${}^{\flat}\gamma : \langle\, oF_0 \,/\, R_0 \,\rangle \Rightarrow \langle\, oF_1 \,/\, R_1 \,\rangle$. We say that $(s)_a$ in oF_0 fires in ${}^{\flat}\gamma$, written $Fires({}^{\flat}\gamma, (s)_a)$, if the underlying open fragment computation of ${}^{\flat}\gamma$ contains an execution step for $(s)_a$. Similarly ${}^{\flat}\gamma$ fires $a \triangleleft v$ (written $Fires({}^{\flat}\gamma, a \triangleleft v)$) if a is external and $a \triangleleft v$ is output in ${}^{\flat}\gamma$, or a is internal and the underlying open fragment transition of ${}^{\flat}\gamma$ contains a delivery step for $a \triangleleft v$.

Definition (*Fires*): $Fires(z, x)$ is the least relation on ${}^{\flat}Cmp_{\mathrm{A}}$ and $\mathbf{(S)_A} \cup \mathbf{A} \triangleleft \mathbf{V}$ such that the following hold.

$$ Fires(\mathtt{e}(a, s, \vec{a}), (s)_a) $$

$$ Fires(\mathtt{d}(a, s, v), a \triangleleft v) $$

$$ Fires(\{\delta_0\,,\ \delta_1\}, x) \quad \text{if} \quad Fires(\delta_0, x) \quad \text{or} \quad Fires(\delta_1, x) $$

$$ Fires(\delta_0\ ;\ \delta_1, x) \quad \text{if} \quad Fires(\delta_0, x) \quad \text{or} \quad Fires(\delta_1, x) $$

$$ Fires(\langle\, \delta \,/\, R \,\rangle, x) \quad \text{if} \quad Fires(\delta, x) $$

$$ Fires(\mathtt{fi}(\delta, R, a \triangleleft v), x) \quad \text{if} \quad Fires(\delta, x) $$

$$ Fires(\mathtt{fo}(\delta, R, a \triangleleft v), x) \quad \text{if} \quad Fires(\delta, x) \quad \text{or} \quad x = a \triangleleft v $$

$$ Fires({}^{\flat}\gamma_0\ ;\ {}^{\flat}\gamma_1, x) \quad \text{if} \quad Fires({}^{\flat}\gamma_0, x) \quad \text{or} \quad Fires({}^{\flat}\gamma_1, x) $$

Note that by definition of open fragment computations, we have $En_{\mathrm{ex}}(a, s)$ in the (**exe**) case and $En_{\mathrm{d}}(a, s, v)$ in the (**del**) case. Also *Fires* is well-defined

on $\sim$-equivalence classes.

A computation path is fair just if a transition that becomes enabled at some stage is either fired at some later stage, or becomes permanently disabled at some later stage.

Definition (fairness):

$$\begin{aligned} Fair(^\flat p) \Leftrightarrow\ & (\forall i \in \mathbf{Nat})(\forall x \in (\mathbf{S})_{\mathbf{A}} \cup \mathbf{A} \triangleleft \mathbf{V})(Enabled(src(^\flat p(i)), x) \Rightarrow \\ & (\exists j \in \mathbf{Nat}) Fires(^\flat p(i+j), x) \vee \\ & (\exists j \in \mathbf{Nat})(\forall k \in \mathbf{Nat}) \neg (Enabled(src(^\flat p(i+j+k)), x))) \end{aligned}$$

Equivalent paths are either both fair or both unfair.

Lemma (equi-fair): If $^\flat p \sim {}^\flat p'$, then $Fair(^\flat p) \Leftrightarrow Fair(^\flat p')$.

5.4 Unflattening $^\flat Rt_{\mathrm{A}}$ to Rt_{A}

We define a rewrite theory map $\overline{(_)} : {}^\flat Rt_{\mathrm{A}} \to Rt_{\mathrm{A}}$. We call this the *unflattening* map. The initial model construction lifts this map to computations (finite and infinite). As noted earlier **oF** is a subsort of **F** in Rt_{A}, and the unflattening map extends this inclusion.

Definition (Unflattening, $\overline{(_)}$):

$$\begin{aligned} & \overline{oF \ / \ R} = oF\lceil R \\ & \overline{\langle\, oF \ / \ R \,\rangle} = \langle\, \overline{oF \ / \ R} \,\rangle = \langle\, oF\lceil R \,\rangle \\ & \overline{\mathtt{e}(a, s, \vec{a})} = \mathtt{e}(a, s) \\ & \overline{\mathtt{d}(a, s, v)} = \mathtt{d}(a, s, v) \\ & \overline{\mathtt{fi}(oF, R, a \triangleleft v)} = \mathtt{i}(oF\lceil R, a \triangleleft v) \\ & \overline{\mathtt{fo}(oF, R, a \triangleleft v)} = \mathtt{o}(oF, R, a \triangleleft v) \end{aligned}$$

By construction the source and target functions commute with the mapping from $^\flat Rt_{\mathrm{A}}$ to Rt_{A}. For example, $src(\overline{^\flat\gamma}) = \overline{src(^\flat\gamma)}$ and $tgt(\overline{^\flat\gamma}) = \overline{tgt(^\flat\gamma)}$. Also equivalent $^\flat Rt_{\mathrm{A}}$ computations (finite or infinite) are mapped to equivalent Rt_{A} computations. Furthermore, the mapping is onto (as a mapping of $\sim$-classes). To see this requires a little work.

Theorem (onto):

(ff) $\overline{^\flat\mathbf{F}} = \mathbf{F}$

(fc) $\overline{^\flat\mathbf{C}} = \mathbf{C}$

(Cmp) $\overline{^\flat Cmp_{\mathrm{A}}} \sim Cmp_{\mathrm{A}}$

(Path) $\overline{^\flat Path_{\mathrm{A}}} \sim Path_{\mathrm{A}}$

Proof : (**ff**) and (**fc**) follow from the (**Open form**) lemma for fragments. (**Cmp**) follows from the lemma (**ocmp**) below. (**Path**) follows from the lemma (**opath**) below. □

Lemma (Open form for finite computations (ocmp)):

(f) If $\tau : F \Rightarrow F'$, with $R = recep(F)$, then we can find δ such that $\tau \sim \overline{\delta \mathbin{/} R}$. We call $\delta \mathbin{/} R$ an *open form* of τ.

(c) If $\gamma : C \Rightarrow C'$, then we can find ${}^{\flat}\gamma$, such that $\gamma \sim \overline{{}^{\flat}\gamma}$. We call ${}^{\flat}\gamma$ an *open form* of γ.

Note that if ${}^{\flat}\tau$ is an open form of τ, then $src({}^{\flat}\tau)$ is a open form of $src(\tau)$. Furthermore, if ${}^{\flat}\tau = \delta \mathbin{/} R : oF \mathbin{/} R \Rightarrow oF' \mathbin{/} R$, and ρ fixes R and $extrn(oF)$, then $\widehat{\rho}(\delta) \mathbin{/} R : \widehat{\rho}(oF) \mathbin{/} R \Rightarrow \widehat{\rho}(oF') \mathbin{/} R$ is also an open form of τ. Thus we may freely pick the addresses of non-receptionist actors (subject to avoiding address conflicts).

Proof (f): The proof of (**f**) is by induction and cases on the structure of τ.

(id) If $\tau = \mathtt{id}[F]$, let $oF \lceil R$ be any open form of F and take $\delta = \mathtt{id}[oF]$.

(e) If $\tau = \mathtt{e}(a, s)$, let $\vec{a}$ be any list of new addresses appropriate for a, s and take $\delta = \mathtt{e}(a, s, \vec{a})$.

(d) If $\tau = \mathtt{d}(a, s, v)$, take $\delta = \tau$.

(r) If $\tau = \tau_0 \lceil R$, where $\tau_0 : F_0 \Rightarrow F_0'$, and $R_0 = recep(F_0)$, let $\delta_0 : oF_0 \Rightarrow oF_0'$ be such that $\delta_0 \lceil R_0 \sim \tau_0$. Take $\delta = \delta_0$ since $\delta_0 \lceil R \sim \delta_0 \lceil R_0 \lceil R$.

(mun) If $\tau = \{\tau_0\,,\ \tau_1\}$, where $\tau_j : F_j \Rightarrow F_j'$, and $R_j = recep(F_j)$, for $j < 2$, let $\delta_j : oF_j \Rightarrow oF_j'$, such that $\tau_j \sim \delta_j \lceil R_j$, and such that $(recep(oF_j') - recep(oF_j)) \cap extrn(oF_{1-j}') = \emptyset$ for $j < 2$. Then take $\delta = \{\delta_0\,,\ \delta_1\}$.

(;) If $\tau = \tau_0 \,;\, \tau_1$, where $\tau_j : F_j \Rightarrow F_j'$, $R = recep(F_j)$, for $j < 2$, and $F_0' = F_1$, let $\delta_j : oF_j \Rightarrow oF_j'$, such that $\tau_j \sim \delta_j \lceil R$ and $oF_0' = oF_1$. Then take $\delta = \delta_0 \,;\, \delta_1$.

$\square_{\mathbf{f}}$

Proof (c):

(f) If $\gamma = \langle\, \tau \,\rangle$, let $\delta \mathbin{/} R$ be an open form for τ, then $\delta \mathbin{/} R$ is an open form of γ.

(i) If $\gamma = \mathtt{i}(\tau, a \lhd v)$, then ${}^{\flat}\gamma = \mathtt{fi}(\delta, R, a \lhd v)$ is an open form for γ for any open form, $\delta \mathbin{/} R$, of τ, such that ${}^{\flat}\gamma$ is well-formed.

(o) If $\gamma = \mathtt{o}(\tau, R, a \lhd v)$, then ${}^{\flat}\gamma = \mathtt{fo}(\delta, R, a \lhd v)$ is an open form for γ for any open form, $\delta \mathbin{/} R'$, of τ, such that ${}^{\flat}\gamma$ is well-formed.

The cases $\gamma = \mathtt{id}[C]$ and $\gamma = \gamma_0 \,;\, \gamma_1$ are similar to the fragment computation case.

$\square_{\mathbf{c}}\ \square_{\mathbf{ocmp}}$

Lemma (Open form for paths (opath)): If $p \in Path_{\mathrm{A}}$, then we can find ${}^{\flat}p : {}^{\flat}Path_{\mathrm{A}}$ such that $\overline{{}^{\flat}p} \sim p$.

Proof : Assume $p \in Path_{\mathrm{A}}$. It is easy to pick ${}^{\flat}\gamma_i$ for $i \in \mathbf{Nat}$ such that $\overline{{}^{\flat}\gamma_i} \sim p(i)$. The trick is to do this so that

$$\langle\, oF_{i+1}' \mathbin{/} R_{i+1}' \,\rangle = tgt({}^{\flat}\gamma_i) = src({}^{\flat}\gamma_{i+1}) = \langle\, oF_{i+1} \mathbin{/} R_{i+1} \,\rangle.$$

Pick ρ_i such that ρ_i fixes R_{i+1} and $extrn(oF_{i+1})$ and $\widehat{\rho}_i(oF_{i+1}') = oF_{i+1}$. Using these renamings we can track any actor from its creation to discover whether it ever becomes a receptionist and if so its exported address. Now we map the address of internal actors of each step to their exported address or to some newly chosen address in $\mathbf{A} - (recep(p) \cup extrn(p))$ (this is one reason for requiring this to set be countable). Then the resulting sequence forms a path

in ${}^{\flat}Path_{\mathrm{A}}$. □

5.5 Observational Fairness

We now define the observably fair paths of Rt_{A}.

Definition (Observational Fairness): The set of observationally fair paths of Rt_{A}, *Ofair* is defined by

$$Ofair = \overline{Fair} = \{\overline{{}^{\flat}p} \mid {}^{\flat}p \in Fair\}$$

The inverse image of *Ofair* may include non-fair paths. To see this, consider the following. Let s be an actor state with the property that $\#new(s) = 0$ and $Ex((s)_a)([\,]) = (s)_a$, and let ${}^{\flat}C = \langle\{(s)_{a_0}, (s)_{a_1}\} / \emptyset\rangle$. Define ${}^{\flat}p^u$, ${}^{\flat}p^f$, p as follows.

$$
\begin{aligned}
&{}^{\flat}p^u(i) = \langle\{\mathtt{e}(a_0, s, [\,]), \mathtt{id}[(s)_{a_1}]\} / \emptyset\rangle\\
&{}^{\flat}p^f(2i) = \langle\{\mathtt{e}(a_0, s, [\,]), \mathtt{id}[(s)_{a_1}]\} / \emptyset\rangle\\
&{}^{\flat}p^f(2i+1) = \langle\{\mathtt{id}[(s)_{a_0}], \mathtt{e}(a_1, s, [\,])\} / \emptyset\rangle\\
&p(i) \sim \langle\{\mathtt{e}(a_0, s), \mathtt{id}[(s)_{a_1}]\}\lceil\emptyset\rangle
\end{aligned}
$$

Then ${}^{\flat}p^u$ is unfair, ${}^{\flat}p^f$ is fair and $\overline{{}^{\flat}p^u} \sim \overline{{}^{\flat}p^f} \sim p$, since

$$p(i) \sim \langle\{\mathtt{e}(a_0, s), \mathtt{id}[(s)_{a_1}]\}\lceil\emptyset\rangle \sim \langle\{\mathtt{id}[(s)_{a_0}], \mathtt{e}(a_1, s)\}\lceil\emptyset\rangle$$

by the axioms (**alpha**) and commutativity of $\{_, _\}$ lifted to computations.

6 Component Algebra with Compositional Semantics

A *computation set* is a set, S, of computations (finite or infinite) with a common source, which we denote by $src(S)$. Components can be given a variety of semantics by mapping them to various sets of (equivalence classes of) admissible computations: finite computations, paths, fair paths, to name a few. We want to extend the parallel composition (aka multiset union) operation on fragments to components and associated computation sets in such a way that the semantics is compositional.

Our approach is to define a restricted and simple notion of parallel composability, $c_0 \# c_1$, and composition, $c_0 \mid c_1$, on computations. Then, composition of computation sets with composable sources is defined as the set of compositions of composable elements (modulo equivalence).

$$(\dagger)\quad S_0 \mid S_1 = \{c \mid (\exists c_0, c_1)(\bigwedge_{j<2} c_j \in S_j) \wedge c_0 \# c_1 \wedge c \sim c_0 \mid c_1\}$$

Although $c_0 \mid c_1$ will lack nice properties such as associativity, when lifted to the computation sets of interest these properties will be recovered.

We want to define parallel composability and composition on components and computations in Rt_{A} such that using (†) we have

$$(\ddagger)\quad \mathcal{X}(C_0) \mid \mathcal{X}(C_1) = \mathcal{X}(C_0 \mid C_1) \qquad \text{if } C_0 \# C_1$$

for $\mathcal{X}$ one of $\{Cmp_{\mathrm{A}}, Path_{\mathrm{A}}, Ofair\}$. We do this by first solving the problem in ${}^{\flat}Rt_{\mathrm{A}}$ and lifting the results using the unflattening map from ${}^{\flat}Rt_{\mathrm{A}}$ to Rt_{A}.

6.1 Parallel Composition in ${}^{\flat}Rt_{\mathrm{A}}$

Composing flat components

$$\langle\, oF_0 \,/\, R_0 \,\rangle \# \langle\, oF_1 \,/\, R_1 \,\rangle \Leftrightarrow$$
$$recep(oF_0) \cap recep(oF_1) = \emptyset \wedge \bigwedge_{j<2} extrn(oF_j) \cap recep(oF_{1-j}) \subseteq R_{1-j}$$
$$\langle\, oF_0 \,/\, R_0 \,\rangle \,|\, \langle\, oF_1 \,/\, R_1 \,\rangle = \langle\, \{oF_0,\ oF_1\} \,/\, R_0 \cup R_1 \,\rangle$$
$$\text{if } \langle\, oF_0 \,/\, R_0 \,\rangle \# \langle\, oF_1 \,/\, R_1 \,\rangle$$

To simplify definitions, we define composability and composition only for computations that are sequences of io-steps. An *io-step* step is a computation of the form $\langle\, \delta \,/\, R \,\rangle$, $\mathtt{fi}(oF, R, a \triangleleft v)$, or $\mathtt{fo}(oF, R, a \triangleleft v)$. ${}^{\flat}Cmp_{\mathrm{ios}}$ is the set of io steps, It is easy to see that for any admissible flat computation (finite or infinite) there is an equivalent flat computation formed from a sequence of io-steps. Thus without loss we can work with paths whose elements are io-steps to define composition. We let ${}^{\flat}Path_{\mathrm{io}} = Path[{}^{\flat}Cmp_{\mathrm{ios}}] \cap {}^{\flat}Path_{\mathrm{A}}$ and we let ${}^{\flat}q$ range over ${}^{\flat}Path_{\mathrm{io}}$. $\overset{\mathrm{io}}{\approx}$ is the equivalence relation on sequences of io-steps obtained by omitting the exchange rules. In the following we make use of the identity convention that oF used where a computation is expected stands for $\mathtt{id}[oF]$.

Composability of io-steps Composability of io-steps is the least symmetric relation such that

(of) $\langle\, \delta_0 \,/\, R_0 \,\rangle \# \langle\, \delta_1 \,/\, R_1 \,\rangle$
if $\langle\, src(\delta_0) \,/\, R_0 \,\rangle \# \langle\, src(\delta_1) \,/\, R_1 \,\rangle \wedge \langle\, tgt(\delta_0) \,/\, R_0 \,\rangle \# \langle\, tgt(\delta_1) \,/\, R_1 \,\rangle$

(in) $\langle\, oF_0 \,/\, R_0 \,\rangle \# \mathtt{fi}(oF_1, R_1, a \triangleleft v)$
if $oF_0 \# oF_1$
$\wedge\ (acq(v) \cap recep(oF_0)) \subseteq R_0$

(out) $\langle\, oF_0 \,/\, R_0 \,\rangle \# \mathtt{fo}(oF_1, R_1, a \triangleleft v)$
if $\langle\, oF_0 \,/\, R_0 \,\rangle \# \langle\, \{oF_1,\ a \triangleleft v\} \,/\, R_0 \,\rangle$
$\wedge\ a \notin recep(oF_0) \wedge acq(v) \cap recep(oF_0) \subseteq R_0$

(io) $\mathtt{fi}(oF_0, R_0, a \triangleleft v) \# \mathtt{fo}(oF_1, R_1, a \triangleleft v)$
if $\langle\, oF_0 \,/\, R_0 \,\rangle \# \langle\, \{oF_1,\ a \triangleleft v\} \,/\, R_1 \,\rangle$

Composing io-steps

(of) $\langle\, \delta_0 \,/\, R_0 \,\rangle \,|\, \langle\, \delta_1 \,/\, R_1 \,\rangle = \langle\, \{\delta_0,\ \delta_1\} \,/\, R_0 \cup R_1 \,\rangle$
$: \langle\, \{src(\delta_0),\ src(\delta_1)\} \,/\, R_0 \cup R_1 \,\rangle \Rightarrow \langle\, \{tgt(\delta_0),\ tgt(\delta_1)\} \,/\, R_0 \cup R_1 \,\rangle$
if $\langle\, \delta_0 \,/\, R_0 \,\rangle \# \langle\, \delta_1 \,/\, R_1 \,\rangle$

(in) $\langle\, oF_0 \,/\, R_0 \,\rangle \,|\, \mathtt{fi}(oF_1, R_1, a \triangleleft v) = \mathtt{fi}(\{oF_0,\ oF_1\}, R_0 \cup R_1, a \triangleleft v)$

$$: \langle \{oF_0,\ oF_1\} \,/\, R_0 \cup R_1 \rangle \Rightarrow \langle \{oF_0,\ oF_1,\ a \lhd v\} \,/\, R_0 \cup R_1 \rangle$$
$$\text{if } \langle oF_0 \,/\, R_0 \rangle \# \mathtt{fi}(oF_1, R_1, a \lhd v)$$

(out) $\langle oF_0 \,/\, R_0 \rangle \,|\, \mathtt{fo}(oF_1, R_1, a \lhd v) = \mathtt{fo}(\{oF_0,\ oF_1\}, R_0 \cup R_1, a \lhd v)$
$$: \langle oF_0 \,/\, R_0 \rangle \,|\, \langle \{oF_1,\ a \lhd v\} \,/\, R_1 \rangle$$
$$\Rightarrow \langle oF_0 \,/\, R_0 \rangle \,|\, \langle oF_1 \,/\, R_1 \cup (acq(v) \cap recep(F_1)) \rangle$$
$$\text{if } \langle oF_0 \,/\, R_0 \rangle \# \mathtt{fo}(oF_1, R_1, a \lhd v)$$

(io) $\mathtt{fi}(oF_0, R_0, a \lhd v) \,|\, \mathtt{fo}(F_1, R_1, a \lhd v)$
$$= \langle \{oF_0,\ oF_1,\ a \lhd v\} \,/\, R_0 \cup R_1 \rangle$$

(sym) ${}^\flat C_0 \,|\, {}^\flat C_1 \sim {}^\flat C_1 \,|\, {}^\flat C_0 \quad \text{if } {}^\flat C_0 \# {}^\flat C_1$

If ${}^\flat\gamma_0$ and ${}^\flat\gamma_1$ are composable io-steps, then ${}^\flat\gamma_0 \,|\, {}^\flat\gamma_1$ is an io-step and $src({}^\flat\gamma_0 \,|\, {}^\flat\gamma_1) = src({}^\flat\gamma_0) \,|\, src({}^\flat\gamma_1)$. However, in the (io) case the receptionists of the target of the composition may be a proper subset of the union of the receptionists of the targets of the composees. This means that paths that are pointwise composable may not compose. For example let ${}^\flat q_0$ and ${}^\flat q_1$ be such that

$${}^\flat q_0(0) = \mathtt{fi}((a_0)_{ad_0}, \{a_0\}, a_0 \lhd a_2)$$
$${}^\flat q_1(0) = \mathtt{fo}(\{(a_1)_{ad_1},\ (a_2)_{ad_2}\}, \{a_1\}, a_0 \lhd a_2).$$
$${}^\flat q_0(1) = \mathtt{id}[tgt({}^\flat q_0(0))]$$
$${}^\flat q_1(1) = \mathtt{fi}(\{(a_1)_{ad_1},\ (a_2)_{ad_2}\}, \{a_1, a_2\}, a_2 \lhd v)$$
$${}^\flat q_j(i+2) = \mathtt{id}[tgt({}^\flat q_j(1))] \quad \text{for} \quad j < 2 \quad \text{and} \quad i \in \mathbf{Nat}$$

Then ${}^\flat q_0(i) \# {}^\flat q_1(i)$ for $i \in \mathbf{Nat}$, but $\lambda i.({}^\flat q_0(i) \,|\, {}^\flat q_1(i))$ is not a path, since letting $oF = \{(a_0)_{ad_0},\ a_0 \lhd a_2,\ (a_1)_{ad_1},\ (a_2)_{ad_2}\}$, we have $tgt({}^\flat q_0(0) \,|\, {}^\flat q_1(0)) = \langle oF \,/\, \{a_0, a_1\} \rangle$, while $src({}^\flat q_0(1))) \,|\, src({}^\flat q_1(1)) = \langle oF \,/\, \{a_0, a_1, a_2\} \rangle$. Thus we have a mismatch of adjacent target and source components and the input to a_2 is not allowed in the composition at stage 1. Care must be taken when extending parallel composability and composition to io-paths avoid such problems. For this purpose we define some auxiliary relations and operations on io-steps: $Ok2R({}^\flat\gamma, R)$ (OK to restrict receptionists), ${}^\flat\gamma \lceil R$ (the result of restriction), $recep({}^\flat q_0, {}^\flat q_1, i)$, (the receptionists at stage i of pointwise composable io-step paths).

$$Ok2R(\langle \delta \,/\, R \rangle, R') \Leftrightarrow Ok2R(\mathtt{fo}(oF, R, a \lhd v), R') \Leftrightarrow R' \subseteq R$$
$$Ok2R(\mathtt{fi}(oF, R, a \lhd v), R') \Leftrightarrow R' \subseteq R \wedge a \in R' \wedge acq(v) \cap recep(oF) \subseteq R'$$
$$\langle \delta \,/\, R \rangle \lceil R' = \langle \delta \,/\, R' \rangle \quad \text{if} \quad Ok2R(\langle \delta \,/\, R \rangle, R')$$
$$\mathtt{fi}(oF, R, a \lhd v) \lceil R' = \mathtt{fi}(oF, R', a \lhd v) \quad \text{if} \quad Ok2R(\mathtt{fi}(oF, R, a \lhd v), R')$$
$$\mathtt{fo}(oF, R, a \lhd v) \lceil R' = \mathtt{fo}(oF, R', a \lhd v) \quad \text{if} \quad Ok2R(\mathtt{fo}(oF, R, a \lhd v), R')$$
$$recep({}^\flat q_0, {}^\flat q_1, 0) = recep(src({}^\flat q_0)) \cup recep(src({}^\flat q_0))$$
$$recep({}^\flat q_0, {}^\flat q_1, i+1) = recep({}^\flat q_0, {}^\flat q_1, i) \cup E$$

$$\text{where} \quad E = \begin{cases} \emptyset & \text{if } {}^\flat q_0(i) \,|\, {}^\flat q_1(i) \text{ is not an output} \\ acq(v) \cap recep(oF) & \text{if } {}^\flat q_0(i) \,|\, {}^\flat q_1(i) = \mathtt{fo}(oF, R, a \lhd v) \end{cases}$$

Composing io-step paths

$$^\flat q_0 \# {}^\flat q_1 \Leftrightarrow (\forall i \in \mathbf{Nat})({}^\flat q_0(i) \# {}^\flat q_1(i) \wedge Ok2R({}^\flat q_0(i) \mid {}^\flat q_1(i), recep({}^\flat q_0, {}^\flat q_1, i)))$$

$$^\flat q_0 \mid {}^\flat q_1 = \lambda i \in \mathbf{Nat}.({}^\flat q_0(i) \mid {}^\flat q_1(i)) \lceil recep({}^\flat q_0, {}^\flat q_1, i) \quad \text{if } {}^\flat q_0 \# {}^\flat q_1$$

Lemma (composing paths):
(ioeP) If $^\flat q_j \in {}^\flat Path_{\mathrm{io}}({}^\flat C_j)$ for $j < 2$ and $^\flat q_0 \# {}^\flat q_1$, then
$^\flat q_0 \mid {}^\flat q_1 \in {}^\flat Path_{\mathrm{io}}({}^\flat C_0 \mid {}^\flat C_1)$
(Fair) If $^\flat q_j \in Fair({}^\flat C_j)$ for $j < 2$ and $^\flat q_0 \# {}^\flat q_1$, then
$^\flat q_0 \mid {}^\flat q_1 \in Fair({}^\flat C_0 \mid {}^\flat C_1)$
Proof : (**IoeP**) is clear from the definitions. To see (**Fair**) we note the following. If $^\flat q = {}^\flat q_0 \mid {}^\flat q_1$ then

$$Enabled(src({}^\flat q(i)), (s)_a) \Leftrightarrow Enabled(src({}^\flat q_0(i)), (s)_a) \vee Enabled(src({}^\flat q_1(i)), (s)_a)$$

$$Fires({}^\flat q(i), (s)_a) \Leftrightarrow Fires({}^\flat q_0(i), (s)_a) \vee Fires({}^\flat q_1(i), (s)_a)$$

$$Enabled(src({}^\flat q(i)), a \triangleleft v)$$
$$\Rightarrow Enabled(src({}^\flat q_0(i)), a \triangleleft v) \vee Enabled(src({}^\flat q_1(i)), a \triangleleft v)$$

$$Fires({}^\flat q_j(i), a \triangleleft v) \Rightarrow Fires({}^\flat q(i), a \triangleleft v) \vee$$
$${}^\flat q_j(i) = \mathtt{fi}(oF_j, R_j, a \triangleleft v) \wedge {}^\flat q_{1-j}(i) = \mathtt{fi}(oF_{1-j}, R_{1-j}, a \triangleleft v)$$

6.2 Decomposing and Compositionality Theorem in $^\flat Rt_{\mathrm{A}}$

Lemma (Decomposition of open computations (docmp)): If $\delta : oF \Rightarrow oF'$ and $oF = \{oF_0, oF_1\}$ then for any $R \subseteq recep(oF)$ we can find sequential compositions of io-steps, $^\flat\gamma_j : \langle oF_j \,/\, R_j \rangle \Rightarrow \langle oF'_j \,/\, R'_j \rangle$ with $R_j = R \cap recep(oF_j)$ for $j < 2$, such that $^\flat\gamma_0 \# {}^\flat\gamma_1$ and $^\flat\gamma_0 \mid {}^\flat\gamma_1 \overset{\mathrm{io}}{\sim} \langle \delta \,/\, R \rangle$.
Proof : Assume $\delta : oF \Rightarrow oF'$, $oF = \{oF_0, oF_1\}$, and $R \subseteq recep(oF)$ and let $R_j = R \cap recep(oF_j)$ for $j < 2$. We find the required $^\flat\gamma_j$ by induction on the construction of δ.
(id) If $\delta = \mathtt{id}[oF]$, take $^\flat\gamma_j = \langle oF_j \,/\, R_j \rangle$.
(e) If $\delta = \mathtt{e}(a, s, \vec{a})$, assume $a \in recep(oF_1)$ (the other case is similar). Thus $oF_1 = (s)_a$, and we may take $^\flat\gamma_0 = \langle \mathtt{id}[\diamond] \,/\, \emptyset \rangle$ and $^\flat\gamma_1 = \langle \delta \,/\, R_1 \rangle$.
(d) If $\delta = \mathtt{d}(a, s, v)$, assume $a \in recep(oF_1)$ (the other case is similar). There are two cases: $oF_0 = \diamond$ and $oF_0 = a \triangleleft v$. In the first case we may take $^\flat\gamma_0 = \langle \mathtt{id}[\diamond] \,/\, \emptyset \rangle$ and $^\flat\gamma_1 = \langle \delta \,/\, R_1 \rangle$. In the second case $R_1 = \{a\}$, and (this is why we need decomposition at the component level) the message must be output by oF_0 and input by oF_1 before the delivery. Thus we take

$$^\flat\gamma_0 = \mathtt{fo}(\diamond, \emptyset, a \triangleleft v) \,;\, \langle \mathtt{id}[\diamond] \,/\, \emptyset \rangle \quad \text{and} \quad {}^\flat\gamma_1 = \mathtt{fi}(oF_1, \{R_1\}, a \triangleleft v) \,;\, \langle \delta \,/\, R_1 \rangle$$

(;) If $\delta = \delta_x \,;\, \delta_y$,then by induction we can find $^\flat\gamma_{x,j} : \langle oF_j \,/\, R_j \rangle \Rightarrow \langle oF'_j \,/\, R'_j \rangle$ such that $^\flat\gamma_{x,0} \mid {}^\flat\gamma_{x,1} \overset{\mathrm{io}}{\sim} \langle \delta_x \,/\, R \rangle$ and $src(\delta_y) = \{oF'_0, oF'_1\}$. Furthermore we can find $^\flat\gamma_{y,j}$ such that $src({}^\flat\gamma_{y,j}) = \langle oF'_j \,/\, R'_j \rangle$, and $^\flat\gamma_{y,0} \mid {}^\flat\gamma_{y,1} \overset{\mathrm{io}}{\sim} \langle \delta_y \,/\, R \rangle$. Taking $^\flat\gamma_j = {}^\flat\gamma_{x,j} \,;\, {}^\flat\gamma_{y,j}$ we are done.

(mun) If $\delta = \{\delta_x\,,\ \delta_y\}$, let $oF_{x,j}$ be such that $oF_x = \{oF_{x,0}\,,\ oF_{x,1}\} = src(\delta_x)$ and $oF_y = \{oF_{y,0}\,,\ oF_{y,1}\} = src(\delta_y)$. Thus $oF_j = \{oF_{x,j}\,,\ oF_{y,j}\}$ for $j < 2$. By induction we can find ${}^\flat\gamma_{x,j}$ and ${}^\flat\gamma_{y,j}$ such that

$$\langle\, \delta_x \,/\, R_x \,\rangle \overset{\text{io}}{\simeq} {}^\flat\gamma_{x,0} \,|\, {}^\flat\gamma_{x,1} \quad \text{and} \quad \langle\, \delta_y \,/\, R_y \,\rangle \overset{\text{io}}{\simeq} {}^\flat\gamma_{y,0} \,|\, {}^\flat\gamma_{y,1}.$$

Take ${}^\flat\gamma_j = ({}^\flat\gamma_{x,j} \,;\, \mathtt{id}[tgt({}^\flat\gamma_{x,j})]) \,|\, (\mathtt{id}[\langle\, oF_{y,j} \,/\, R_{y,j} \,\rangle] \,;\, {}^\flat\gamma_{y,j})$.

□

Lemma (Decomposition of io-steps (dios)): If ${}^\flat\gamma : {}^\flat C \Rightarrow {}^\flat C'$ is an io-step, and ${}^\flat C = {}^\flat C_0 \,|\, {}^\flat C_1$, then we can find sequential compositions of io-steps, ${}^\flat\gamma_j$, such that $src({}^\flat\gamma_j) = {}^\flat C_j$ for $j < 2$, ${}^\flat\gamma_0 \,\#\, {}^\flat\gamma_1$, and ${}^\flat\gamma_0 \,|\, {}^\flat\gamma_1 \overset{\text{io}}{\simeq} {}^\flat\gamma$.

Proof :

(of) If ${}^\flat\gamma = \langle\, \delta \,/\, R \,\rangle$ then we are done by (**docmp**).

(in) If ${}^\flat\gamma = \mathtt{fi}(oF, R, a \triangleleft v)$, then assuming $a \in recep({}^\flat C_1)$ (the other case is similar), let ${}^\flat C_1 = \langle\, oF_1 \,/\, R_1 \,\rangle$ and take ${}^\flat\gamma_0 = \mathtt{id}[{}^\flat C_0]$, ${}^\flat\gamma_1 = \mathtt{fi}(oF_1, R_1, a \triangleleft v)$.

(out) If ${}^\flat\gamma = \mathtt{fo}(oF, R, a \triangleleft v)$, then ${}^\flat C = \langle\, \{oF\,,\ a \triangleleft v\} \,/\, R \,\rangle$. Assuming $a \triangleleft v \in {}^\flat C_1$, we may write ${}^\flat C_1 = \langle\, \{oF_1\,,\ a \triangleleft v\} \,/\, R_1 \,\rangle$ where $oF = \{oF_0\,,\ oF_1\}$, $R = R_0 \cup R_1$, and $acq(v) \cap recep(oF) - R = acq(v) \cap recep(oF_1) - R_1$. Thus we may take ${}^\flat\gamma_0 = \mathtt{id}[{}^\flat C_0]$, ${}^\flat\gamma_1 = \mathtt{fo}(oF_1, R_1, a \triangleleft v)$.

□**dios**

Lemma (Decomposition of io-step paths): Let ${}^\flat q \in {}^\flat Path_{\text{io}}(\langle\, oF \,/\, R \,\rangle)$, where $\langle\, oF \,/\, R \,\rangle = \langle\, oF_0 \,/\, R_0 \,\rangle \,|\, \langle\, oF_1 \,/\, R_1 \,\rangle$. Then we can find ${}^\flat q_j \in {}^\flat Path_{\text{io}}(\langle\, oF_j \,/\, R_j \,\rangle)$ such that ${}^\flat q_0 \,\#\, {}^\flat q_1$ and ${}^\flat q \overset{\text{io}}{\simeq} {}^\flat q_0 \,|\, {}^\flat q_1$. Furthermore if ${}^\flat q$ is fair then we can pick ${}^\flat q_j$ to be fair.

Proof : By (**dios**) we can pick ${}^\flat\gamma_{j,i}$ for $j < 2$ and $i \in \mathbf{Nat}$ such that ${}^\flat\gamma_{0,i} \,\#\, {}^\flat\gamma_{1,i}$, $({}^\flat\gamma_{0,i}, {}^\flat\gamma_{1,i})$ decomposes ${}^\flat q(0)$ according to $(\langle\, oF_0 \,/\, R_0 \,\rangle, \langle\, oF_1 \,/\, R_1 \,\rangle)$, and $({}^\flat\gamma_{0,i+1}, {}^\flat\gamma_{1,i+1})$ decomposes ${}^\flat q(i{+}1)$ according to $(tgt({}^\flat\gamma_{0,i}), tgt({}^\flat\gamma_{1,i}))$. To insure fairness, it suffices to insert input/output steps, as done in (**docmp**), for each message created in one subcomponent and targeted to an actor in the other.

□

Theorem (Compositionality in ${}^\flat Rt_{\text{A}}$):

$${}^\flat\mathcal{X}({}^\flat C_0) \,|\, {}^\flat\mathcal{X}({}^\flat C_1) = {}^\flat\mathcal{X}({}^\flat C_0 \,|\, {}^\flat C_1) \qquad \text{if } {}^\flat C_0 \,\#\, {}^\flat C_1$$

for ${}^\flat\mathcal{X}$ one of $\{{}^\flat Cmp_{\text{A}}, {}^\flat Path_{\text{A}}, Fair\}$.

Proof : $\subseteq$ is by (**composing paths**) and $\supseteq$ is by (**decomposition of io-step paths**). □

6.3 Lifting Compositionality to Rt_{A}

Composing Components

$$C_0 \,\#\, C_1 \;\Leftrightarrow\; (\exists {}^\flat C_0, {}^\flat C_1)(\bigwedge_{j<2} \overline{{}^\flat C_j} = C_j \wedge {}^\flat C_0 \,\#\, {}^\flat C_1)$$

$$C_0 \,|\, C_1 = \overline{{}^\flat C_0 \,|\, {}^\flat C_1} \quad \text{if} \quad \bigwedge_{j<2} \overline{{}^\flat C_j} = C_j \wedge {}^\flat C_0 \,\#\, {}^\flat C_1$$

Lemma (Component composition):

$$\langle F_0 \rangle \# \langle F_1 \rangle \Leftrightarrow recep(F_0) \cap recep(F_1) = \emptyset$$
$$\langle F_0 \rangle \mid \langle F_1 \rangle = \langle \{F_0, F_1\} \rangle \quad \text{if } \langle F_0 \rangle \# \langle F_1 \rangle$$

Composing computation sets Let ${}^\flat S_j$ be computation sets in ${}^\flat Rt_A$ with sources ${}^\flat C_j$ and let $S_j = \overline{{}^\flat S_j}$ be the images in Rt_A for $j < 2$. Define

$$S_0 \mid S_1 = \overline{{}^\flat S_0 \mid {}^\flat S_1} \quad \text{if } {}^\flat C_0 \# {}^\flat C_1$$

Thus $S_0 \mid S_1 = \{c \mid (\exists {}^\flat c_0 \in {}^\flat S_0, {}^\flat c_1 \in {}^\flat S_1)({}^\flat c_0 \# {}^\flat c_1 \wedge c \sim \overline{{}^\flat c_0 \mid {}^\flat c_1})\}$

Theorem (Compositionality in Rt_A):

$$\mathcal{X}(C_0) \mid \mathcal{X}(C_1) = \mathcal{X}(C_0 \mid C_1) \qquad \text{if } C_0 \# C_1$$

for $\mathcal{X}$ one of $\{Cmp_A, Path_A, Ofair\}$.

Proof : By compositionality in ${}^\flat Rt_A$. Since the rewriting map from ${}^\flat Rt_A$ to Rt_A sends ${}^\flat Cmp_A$ onto Cmp_A, ${}^\flat Path_A$ onto $Path_A$ and $Fair$ onto $Ofair$. □

An alternative to the above definition of $S_0 \mid S_1$ in Rt_A would be to lift composition of io-steps and io-step sequences to the images of these computation sets under the unflattening map and then use the general definition of path set composition. This would give the same result and would be useful if we want to compute compositions in Rt_A.

7 Future Directions

In this paper we have presented a semantic framework for actor systems based on rewriting logic. This framework accounts for fairness and provides a variety of semantics for components of actor systems that have good composability properties. There are many directions for future work.

In the development of the actor rewriting semantics we have made a number of modifications of and extensions to the standard initial model construction of rewriting logic. These involved picking out subsets of computations, adding infinite computations, and using alternate notions of equivalence on computations. These particular variants seem well-behaved and have a number of possible explanations using concepts such as strategies [5], membership algebras [18] at the model level, and co-limits. There are other examples of such variants, for example in the studies of Petri Net semantics [8,9,20], and we conjecture that many future applications of rewriting will find such variants useful. Thus it seems important to study the model theory of rewriting logic further and to develop formal criteria for characterizing 'nice' variations and to define general operations for constructing them.

Fairness has been treated simply by defining a predicate that picks out the fair computations. There are a number of treatments of fair semantics in the process algebra literature ([7,10,11,14,13] to mention a few) and it will be interesting to investigate whether any of these approaches work for the actor model and how they fit into the rewriting logic framework.

The interaction semantics of a component derived from the Actor rewriting theory in [21] is, on the surface, an amorphous set of `i/o` traces. In fact there is much structure implicit in an interaction set, and it is important to make this structure explicit to facilitate specifying and reasoning about components. We expect the structure of the underlying rewrite terms will be useful in carrying out this task.

There are a number of interesting elaborations of the basic actor model and corresponding elaborations of the AAS/rewriting framework to consider. Among these are: making distribution explicit by introducing locations; modeling mobility of actors; adding regions of synchrony or timed interactions, thus providing a model with both local synchronous and distributed asynchronous computation; modeling reflective actor computation using meta relations or meta actors and reflective interactions.

Last, but not least, an important future project is defining Maude modules for the construction of an actor rewrite theory from a module describing a particular Abstract Actor Structure. This will provide a tool for prototyping actor systems and reasoning about finite aspects of actor computation.

Acknowledgements

The author would like to thank José Meseguer and Manuel Clavel for many helpful discussions of rewriting logic, and her collaborators in actor research, Gul Agha, Ian Mason, and Scott Smith for many fruitful discussions about actor semantics. Special thanks to Ian Mason and Scott Smith for careful reading of earlier versions. This research was partially supported by ARPA grant NAVY N00014-94-1-0775, ONR grant N00014-94-1-0857, NSF grant CCR-9312580, and ARPA/SRI subcontract C-Q0483.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[2] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 1997. to appear.

[4] Henry G. Baker and Carl Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, August 1977.

[5] M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In *Reflection'96*, 1996.

[6] W. D. Clinger. Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[7] G. Costa and C. Stirling. A fair calculus of communicating systems. *Acta Informatica*, 21:417–441, 1984.

[8] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing net computations and processes. In *Fourth Annual Symposium on Logic in Computer Science*, pages 175–185. IEEE, 1989.

[9] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing the algebra of net computations and processes. Technical Report SRI-CSL-90-12, SRI International, Computer Science Laboratory, November 1990. To appear in *Acta Informatica.*

[10] N. Francez. *Fairness.* Springer-Verlag, 1986.

[11] M. C. B. Hennessey. An algebraic theory of fair asynchronous communicating processes. *Theoretical Computer Science*, 49:121–143, 1987.

[12] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[13] M. Kwiatkowska and M. Huth. Finite but unbounded delay in synchronous ccs. Technical Report Technical Report CSR-96-4, Universtiy of Birmingham, February 1996. To appear in TFM'96.

[14] M. Kwiatkowska and M. Huth. The semantics for fair recursion with divergence. Technical Report Technical Report CSR-96-4, Universtiy of Birmingham, April 1996.

[15] J. Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02, SRI International, Computer Science Laboratory, February 1990. Revised June 1990.

[16] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[17] J. Meseguer. A logical theory of concurrent objects and its realization in the maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency.* The MIT Press, 1993.

[18] J. Meseguer. Membership algebras, 1996. Talk given at Dagstuhl Seminar on Specification and Semantics, July 1996.

[19] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Concur96*, 1996.

[20] V. Sassone, J. Meseguer, and U. Montanari. Inductive completion of monoidal categories and infinite net computations. Submitted for publication.

[21] C. L. Talcott. Interaction semantics for components of distributed systems. In *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'96*, 1996.

Electronic Notes in Theoretical Computer Science 4 (1996)

Object-Oriented Specifications of Distributed Systems in the μ-Calculus and Maude

Ulrike Lechner [1]

Fakultät für Mathematik und Informatik
Universität Passau
D-94030 Passau, Germany

Abstract

We refine an abstract property-oriented specification in the μ-calculus to a specification in Maude. As an intermediate step, we use a structured specification in the μ-calculus blended with propositions on states appropriate for object-oriented specification. We use the loose approach in refinement and refine data types as well as behavior. Throughout, our example is the bounded buffer.

1 Introduction

Specification languages provide a level of abstraction from implementation details in the design of complex systems. Specifications are property-oriented descriptions, while programs are executable descriptions of an algorithm. We use two specification formalisms, the μ-calculus [5,19,29] and Maude [25,28] for the object-oriented specification of concurrent systems.

Maude has been developed especially for the object-oriented specification of concurrent systems. The rewriting calculus and the underlying rewriting logic make Maude to very powerful and general specification language [25,27]. Maudes advantages are its object model, its ability to combine the two paradigms of inheritance and concurrency in a sensible way [21,26] and, particularly, its abstract way of specifying synchronization and communication between objects [22]. But, on the other hand, Maude's semantics is operational, and thus not really property-oriented, and the transition rules specifying the behavior of objects are not powerful enough to express, e.g., safety properties. The μ-calculus is property-oriented, i.e., it is able to express safety and liveness properties of the behavior of a concurrent system [5,19,29]. We enrich the μ-calculus with basic propositions on states that make it possible to reason also about the properties of states, not only about the properties of the dynamic behavior.

[1] The author is supported by the DFG under project OSIDRIS and received travel support from the ARC program of the DAAD.

© *1996 Elsevier Science B. V.*

There exist several approaches to specify or describe objects, especially objects in a concurrent setting and to give a formal semantics. We follow the basic concepts of [16,17,31], where the semantics of a class specification is given by a coalgebraic construction. Such a coalgebra specifies the observations one can make of an object or a class, it does not specify how a class is built up and which data it contains. (In the algebraic approach constructors describe which properties an object has, which data it contains [6]). This coalgebraic construction reflects the idea of an observable properties rather than the properties of the implementation of a class. We adopt this idea and use the μ-calculus with greatest fixpoint as our construct of specification.

We use three levels of specification with different degrees of abstraction:

(i) At the *abstract* level, we use the language of modal μ-formulas for specification. A typical specification would describe, e.g., the sequences of messages an object or a collection of objects accepts. Invariants on states restrict the transition system such that all states obey certain wellformedness conditions.

(ii) At the *intermediate* level, we use again the language of μ-formulas. The propositions on states we introduce describe the states of the objects. At this level, the formulas have a very rigid structure. The behavior of the objects belonging to a common class is a conjunction of five formulas specifying (1) that objects are persistent (2) the consistent state of an object (3) the synchronization code determining which messages are accepted depending on the local state of an object (4) the state changes of the objects and (5) the answer messages generated.

(iii) At the *concrete* level, we use Maude as our specification language. At this point, Maude itself provides us with a choice of the degree of abstraction, using, e.g., implicit synchronous or explicit asynchronous communication.

This refinement from an abstract to a concrete specification is reflected at the semantic level by the loose approach to refinement. In this approach, the semantics of a specification is the set of all models that satisfy all formulas of the specification. In each refinement step, the set of models of the specification becomes smaller. The last, most concrete step in such a refinement typically yields a singleton set of models – a program. In this paper all specifications are at a very high level of abstraction: our abstract, property-oriented specification language is the μ-calculus, our concrete, executable language the specification language Maude.

In the process of refinement the properties that shall be preserved determine the kind of relation between the different levels of specification. Particularly the refinement of concurrent systems offers a large variety of relations between transition systems. From [7,20,23] it is known that only a bisimulation relation between finite transition systems preserves all properties, which the μ-calculus can express. Coarser relations between transition systems preserve only certain classes of μ-formulas.

The paper is organized as follows: In Sect. 2 we give an introduction to Maude and the μ-calculus. The refinement relation is defined in Sect. 3. In

Sect. 4 we give specifications at different levels of abstraction and relate the three different levels. Throughout, our example is the bounded buffer. In Sect. 5 we relate our approach of specification and refinement to other work.

Our mathematical notation follows Dijkstra [10]. Quantification over a dummy variable x is written $(Q\,x\,:\,R(x)\,:\,P(x))$. Q is the quantifier, R a predicate in x representing the range of the dummy and P a term that depends on x. E.g., $(\cup x : x \in X \wedge \phi(x) : x)$ is the set of all elements of X for which $\phi(x)$ holds. Formal logical deductions are written:

$$\begin{array}{ll} & \mathit{formula}_1 \\ \text{op} & \quad \{ \text{ comment explaining the validity of this relation } \} \\ & \mathit{formula}_2 \end{array}$$

2 The specification languages

2.1 Maude

Maude [25–28] is an object-oriented specification language for the specification of distributed systems. In this section we assume prior knowledge of Maude and explain only the aspects of Maude relevant in our work. Let us give the specification of a bounded buffer, BDBUFFER, and explain it later.

```
omod BDBUFFER is
  protecting OIDLIST .
  extending CONFIGURATION .

  class BdBuffer | in: Nat, out: Nat, max: Nat, cont: OIdList .
  msg (to _ : get)  : OId -> Msg .
  msg (answer to get is _ ) :  OId -> Msg .
  msg (to _ : put _ ) : OId OId -> Msg .

  vars B U E : OId .
  vars I O M : Nat .
  var L : OIdList .

  [get] rl  (to B: get)
            < B:BdBuffer | in:I, out:O, max:M, cont:L E >
            => < B:BdBuffer | out:O+1, cont:L >
               (answer to get is E)
            if I - O > 0 and I - O =< M
               and length(L E) = I - O .

  [put] rl  (to B: put E)
            < B:BdBuffer | in:I, out:O, max:M, cont:L >
            => < B:BdBuffer | in:I+1, cont:E L >
            if I - O < M and  I-O >= 0
               and length(L) = I - O .
endom
```

In the specification BDBUFFER we declare one class, BdBuffer with four attributes. The behavior of a bounded buffer is specified by two transition rules, with label get and put. They specify whether and how a bounded buffer reacts to a put or get message: If the pattern at the left hand side of a rule matches a buffer and a message in a configuration, and if the precondition of the rule holds, then a state transition *may* happen such that the message is removed from the configuration, the buffer changes its state, i.e., the values of attributes, according to the rule and, possibly, an answer message is generated and part of the resulting configuration.

BDBUFFER imports two specifications: CONFIGURATION contains the basic data types like objects, messages and configurations (see, e.g., [28]) and OIDLIST the specification of lists of objects identifiers (see App. A).

Important for us is that Maude employs asynchronous message passing and, thus, the rewrite rules specify the possible state transitions that *may* happen. Important is also that Maude abstracts from the implementations of methods, what we specify here is the communication between objects and the state changes of objects and the overall system and not how the state changes are performed.

A rewriting calculus applies these rules to configurations. We use as in [21] a simplified version of a rewriting calculus. Let us introduce some notation. A *specification* $Sp = (\Sigma, E, T)$ consists of a *signature* Σ, a set of *equations* E and a set of *transition rules* T. A signature $\Sigma = (S, C, \leq, F, M)$ consists of a set of (ordinary) sort names S, a set of class names C, a subclass relation $\leq$, a set of function symbols F and a set of messages M. $T(\Sigma, X)$ denotes the terms with variables from X of a signature Σ. We use Cf as an abbreviation for Configuration, the sort of the states.

The *rewriting calculus*, given below in three rules, defines Maude's semantics in the form of a *transition system.*[2] In the following, let m, m' denote messages, a_i attribute names, v_i and w_i values, o_i object identifiers, C_i, C_i', D_i and D_i' class identifiers, $atts_i$ sets of pairs of attributes together with their variables, and σ a substitution. An expression e adorned with an overbar, $\overline{e}$ stands for a set whose elements are of the form e (with the exception that $\overline{m}$ is a multiset of messages).

A transition

$$\frac{\overline{m}[\sigma] \quad \overline{< \sigma(o_i) : D_i \mid \overline{a_i : v_i}[\sigma],\ atts_i >}}{\overline{m'}[\sigma] \quad \overline{< \sigma(o_i) : D_i' \mid \overline{a_i : w_i}[\sigma],\ atts_i >}} \qquad \text{(Inst)}$$

is possible if T contains a transition rule (in which all attributes of classes C_i together with their values are stated)

[2] In contrast to [25] we neither have a reflexivity nor a transitivity rule nor parallel composition in the calculus. The rule (Emb) is weaker than the replacement rule in the original calculus; the replacement rule could be obtained by (Emb), (Equ), and a transitivity rule.

$[R]$ $\overline{m}$

$\overline{< o_i : C_i \mid \overline{a_i : v_i} >}$

$\Rightarrow \overline{< o_i : C'_i \mid \overline{a_i : w_i} >}$

$\overline{m'}$

and a substitution $\sigma : Vars \to T(\Sigma, X)$,
where $D_i \leq C_i$ and

$$D'_i = \begin{cases} D_i, \text{ if } C_i = C'_i \\ C'_i \text{ else} \end{cases}$$

In the case of a conditional transition rule of the form:

$m'_1 o'_{i_1} \ldots o'_{i_n} \to o'_{j_1} \ldots o'_{j_m} m'_2 \ldots m'_n$ if $p_1 \wedge \ldots \wedge p_k$

(with equations or transitions $p_1, \ldots, p_k$) we require additionally that all $p_i[\sigma]$ are derivable. We need two more rules: (Emb) embeds the left-hand and the right-hand side of a transition into a configuration, containing objects and messages not changed by the transition and (Equ) makes the transition relation compatible with equations. Let c, d, c', d' and h be configurations and let $=_E$ denote equality modulo equations in the set E:

$c\ h \to d\ h$ if $c \to d$ (Emb)

$c' \to d'$ if $c \to d$ and $c =_E c', d =_E d'$ (Equ)

A structure (A, R) is an initial model of specification $Sp = (\Sigma, E, T)$, written (A,R) = I(Sp), iff

- A is an order sorted Σ-algebra and A is the initial model of (Σ, E) [12].
- R is a relation, $R \subseteq A_{\text{Cf}} \times A_{\text{Cf}}$ such that $(c^A, d^A) \in R$ iff $Sp \vdash c \to d$ where the rules of the calculus are substitution rules, (Inst), (Emb) and (Equ).

Later in this paper we use labeled transitions systems. A transition is computed from one application of the rule (Inst) and applications of (Emb) and (Equ). The label is $m[\sigma]$, where m is the multiset of messages part of the left-hand side of the transition rule and σ the substitution applied by (Inst). Analogously we adapt definition of the relation $R \subseteq A_{\text{Cf}} \times A_{\text{Msg}} \times A_{\text{Cf}}$ where $(c^A, m^A, d^A) \in R$ iff $Sp \vdash c \xrightarrow{m} d$. We also use the notation R_{m^A} for the subset of R with label m^A. (t^A is the representation of a ground term t in algebra A.)

2.2 The μ-calculus

The μ-calculus is used to reason about state transition systems at a property-oriented level. The language of μ-formulas consists of propositions, for reasoning about states, the modal connectives, quantifiers and fixpoint operators.

Our language of propositions for a specification is given by the grammar:

$$p ::= \texttt{tt} \mid \texttt{ff} \mid \neg p \mid \text{“}o\text{”} \mid \text{“}m\text{”}$$

where o, respectively m, is a term over a signature Σ representing an object respectively a message. The double quotes around an object or message repre-

sent the proposition "this object exists" or "this message exists", respectively. E.g., state C satisfies "<B1:BdBuffer|in:1>" if one of its elements is an object with object identifier B1 belonging to class BdBuffer (which includes all subclasses of BdBuffer) whose value of attribute in is equal to 1. Note that the use of negation is restricted to basic propositions.

Let p be a proposition. We define the formulas of the *modal μ-calculus* over a set of basic propositions of signature Σ as follows:

$$\begin{aligned}\phi ::= & \; p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\ & \mid \langle L \rangle \phi \mid [L]\phi \\ & \mid (\nu X : X \in \Phi : \phi) \mid (\mu X : X \in \Phi : \phi)\end{aligned}$$

L is a set of labels. $[L]\phi$ and $\langle L \rangle \phi$ are the labeled modal connectives. $[L]$ is called the box operator, $\langle L \rangle$ is called the diamond operator. Intuitively, $[L]\phi$ holds, if ϕ holds immediately after all transitions with labels in L. Dually, $\langle L \rangle \phi$ holds, if there is a transition with a label in L such that ϕ holds immediately afterwards. We use $\langle - \rangle$ and $[-]$ as abbreviations for modal connectives with the label set of all possible labels.

ν is the greatest fixpoint operator used, typically, for invariant (safety, "always") properties. μ is the least fixpoint operator used, typically, for variant (liveness, "sometime") properties.

We are interested in the truth of formulas in a structure (A, R) which is a model of a Maude specification $Sp = (\Sigma, E, T)$. Let $v : \mathrm{VAR} \to T(\Sigma)$ be a valuation and let $v^\star$ denote the canonical extension of v to an interpretation function for terms. Let $X := C$ be a valuation where C is assigned to X and $w + v$ a valuation such that $w + v(X) = w(X)$ if $X \in \mathrm{dom}(w)$ and $v(X)$ if $X \notin \mathrm{dom}(w)$. Let t^A denote the representation in algebra A of a ground term t. Let $FV(f)$ denote the free variables of formula f. We define $|\phi|_{(A,R),v}$ as the set of all states for which ϕ under the valuation v holds.

We define truth of formulas of the μ-calculus with respect to a state (or configuration) $C \in A_{\mathrm{Cf}}$ in Fig. 1.

We define for a μ-specification $\Phi = \{\phi_1, \ldots, \phi_n\}$ with μ-formulas over a algebraic specification (Σ, E) the class of models of Φ, $\mathrm{MOD}(\Phi)$, as the set of structures (A, R) where A is a model of (Σ, E) and $(A, R) \models \phi_1 \wedge \ldots \wedge \phi_n$.

3 Refinement relation

In the process of stepwise refinement an abstract (requirement) specification is transformed into a concrete specification, which might be a program [32]. We use the so-called loose approach to refinement: We consider the class of all models as the semantics of a specification. In the process of stepwise refinement, implementation details of data types and algorithms are added to the specification. This reduces the class of models (in several steps) to a singleton set – a program.

$(A,R),C,v \models$ "<o:X|$\overline{\texttt{a:v}}$>" iff $v^*(\texttt{<o:X|}\overline{\texttt{a:v}},\overline{\texttt{b:w}}\texttt{>}) \in C$ for some $\overline{\texttt{w}}$ and $a \cup b$ are the attributes of class X

$(A,R),C,v \models$ "m" iff $v^*(m) \in C$

$(A,R),C,v \models \neg\phi$ iff $(A,R),C,v \not\models \phi$

$(A,R),C,v \models \phi_1 \wedge \phi_2$ iff $(A,R),C,v \models \phi_1$ and $(A,R),C,v \models \phi_2$

$(A,R),C,v \models \phi_1 \vee \phi_2$ iff $(A,R),C,v \models \phi_1$ or $(A,R),C,v \models \phi_2$

$(A,R),C,v \models \langle L\rangle\phi$ iff for some $l \in L$, $(C,C') \in R_{v^*(l)}$ and $(A,R),C',v \models \phi$

$(A,R),C,v \models [L]\phi$ iff $l \in L$ and $(C,C') \in R_{v^*(l)}$ implies $(A,R),C',v \models \phi$

$(A,R),C,v \models (\mu X : X \in \Phi : \phi)$ iff

$$C \in (\cap C' : C' \subseteq \Phi^A, |\phi|_{(A,R),X:=C'+v} \subseteq C' : C')$$

$(A,R),C,v \models (\nu X : X \in \Phi : \phi)$ iff

$$C \in (\cup C' : C' \subseteq \Phi^A, |\phi|_{(A,R),X:=C'+v} \supseteq C' : C')$$

We define

$(A,R),C \models \phi$ iff for all $v : FV(\phi) \to A_s$ holds $(A,R),C,v \models \phi$

$(A,R) \models \phi$ iff for all $C \in A_{\mathrm{Cf}}$ holds $(A,R),C \models \phi$

Fig. 1. Truth of μ-formulas

Definition 3.1 Let Σ be signature and (Σ,E) be an (algebraic) specification.

Let $\Phi = \{\phi_1,\ldots\phi_n\}$ and $\Phi' = \{\phi'_1,\ldots\phi'_{n'}\}$ be μ-specifications.

Φ is refined by Φ' written $\Phi \rightsquigarrow \Phi'$ iff $\phi'_1 \wedge \ldots \wedge \phi'_{n'} \implies \phi_1 \wedge \ldots \wedge \phi_n$.

Let $\Phi' = \{\phi'_1,\ldots\phi'_{n'}\}$ be a μ-specification over (Σ,E) and $Sp = (\Sigma,E,T)$ a Maude specification.

Φ' is refined by Sp, written $\Phi' \rightsquigarrow Sp$ iff $I(Sp) \models \phi'_1 \wedge \ldots \wedge \phi'_{n'}$

Let us briefly explain the relations between the specifications, particularly between specifications in different languages. The specifications have a basic signature andalso basic data types, specified by equations, in common. Thus the order-sorted algebra A and, in particular, the states, i.e., the terms of sort Configuration (abbreviated Cf) are the same. Different is the level of abstraction in the language for specifying the behavior of objects but common to these languages are the transition system: the semantics of a Maude specification is a transition system and the μ-properties are verified for the transition system.

4 The specifications

We give three specifications, Φ_A, Φ_M, Sp_C of a bounded buffer, each at a different level of abstraction, such that

$$\Phi_A \rightsquigarrow \Phi_M \rightsquigarrow Sp_C$$

4.1 The abstract level – Invariants and μ-calculus

Relevant at the most abstract level are –for us– only two issues:

- *What is a bounded buffer?*
- *How does a bounded buffer behave?*

The question "What is a bounded buffer?" can be answered by specifying a property each bounded buffer has to satisfy:

$$\mathit{State}(\mathtt{B}) = \text{“<B:BdBuffer>”} \implies 0 \leq \mathtt{length(B.cont)} \leq \mathtt{B.max}$$

The number of elements stored in a buffer is `length(B.cont)`. It is not larger than `B.max`, the maximal number of elements of a buffer.

At this abstract level, we do not give the implementation of the state, but, instead, we require that a buffer has certain properties: it is able to determine the length of its contents and it stores only the maximum number of elements.

"What is a bounded buffer?" is answered by specifying properties of the state. This is usually not done in the μ-calculus. The specification of "How does a bounded buffer behave?" is answered by giving properties that do not involve the state but the interactions of objects via messages. For specifying this aspect we use the modal μ-calculus.

So, a general approach to answering "what is ... "? is to give basic properties and functions (`cont`, `max`, `length`) as well as invariants for objects and configurations (here we have only an invariant for one object, *State*). The answer to "how does ... behave?" gives possible actions (or messages) of an object together with their allowed or required (sequences of) actions.

If the predicate *State* holds in some state for a buffer `B`, then it holds in all subsequent states for `B`:

$$\phi_1(\mathtt{B}) = \mathit{State}(\mathtt{B}) \implies (\nu X :: \mathit{State}(\mathtt{B}) \wedge [-]X)$$

The notation $[-]$ is an abbreviation for $[L]$, where L is the set of all possible actions. We do not care what happens with buffers that are in an inconsistent state. ϕ_1 ensures that a consistent buffer remains consistent.A bounded buffer accepts a `put` or a `get` and possibly both messages:

$$\begin{aligned}\phi_2(\mathtt{B}) = \text{“<B:BdBuffer>”} \implies \\ (\nu X :: (\wedge \mathtt{E} : \mathtt{E} \in \mathtt{OId} : \\ (\langle \mathtt{(to\ B:\ put\ E)} \rangle X \vee \langle \mathtt{(to\ B:\ get)} \rangle X)))\end{aligned}$$

After an element has been put into the buffer, there is a sequence of `get` messages such that an `answer` carries the element. The result of a `get` message

is an answer message that is part of the configuration waiting to be processed from –maybe– the object that sent the `get` message. At this point we apply already the asynchronous message passing mechanism of Maude to make refinement later feasible.

$$\begin{aligned}\phi_3(\texttt{B}) = \text{“}&\texttt{<B:BdBuffer>}\text{”} \implies \\ &(\nu X :: (\wedge \texttt{E} : \texttt{E} \in \texttt{OId} : \\ &\quad [\texttt{(to B: put E)}] \\ &\quad (\mu Y :: \langle \texttt{(to B: get)} \rangle \\ &\quad\quad (Y \vee \text{“}\texttt{(answer to get is E)}\text{”})) \wedge X))\end{aligned}$$

After storing two elements in a buffer, the element stored first is retrieved before the one stored second (FIFO):

$$\begin{aligned}\phi_4(\texttt{B}) = \text{“}&\texttt{<B:BdBuffer>}\text{”} \implies \\ &(\nu X :: (\wedge \texttt{E1}, \texttt{E2} : \texttt{E1}, \texttt{E2} \in \texttt{OId} : \\ &\quad [\texttt{(to B: put E1)}][\texttt{(to B: put E2)}] \\ &\quad (\mu Y :: \langle \texttt{(to B: get)} \rangle \\ &\quad\quad (Y \vee (\text{“}\texttt{(answer to get is E1)}\text{”} \\ &\quad\quad\quad \wedge \langle \texttt{(to B: get)} \rangle \\ &\quad\quad\quad\quad \text{“}\texttt{(answer to get is E2)}\text{”}))) \wedge X))\end{aligned}$$

The abstract specification of a bounded buffer is $\Phi_A = \{\phi_1, \phi_2, \phi_3, \phi_4\}$. This set of formulas is just one suggestion to specify the behavior of a bounded buffer. Naturally one could think about entirely different sets of formulas.

4.2 The intermediate level – Structured μ-calculus

At this level of abstraction, we make the decision about the implementation of the internal state of a buffer, namely that the internal state is represented by a proposition “`< B:BdBuffer|in:I,out:O,max:M,cont:L >`”. Implicitly, also the object model manifests itself in the structure of the formulas.

Five formulas determine the behavior of a class. Each formula corresponds to a certain view. We have two internal views which specify consistent states and the state changes induced by the object, we have a property stating that objects are persistent and views for two interfaces: the answer messages that are produced and the link between the incoming messages and the state changes.

Let us give the formula schemata and explain them when applied to the specification of the bounded buffer.

Definition 4.1 Let `C` be a classname and `atts` resp. `atts'` denote the attributes with their values of class `C`. Let $SI(\texttt{<B:C|atts}_i\texttt{>})$, $\phi_i(\texttt{<B:C|atts}_i\texttt{>})$ and $\psi_i(\texttt{<B:C|atts}_i\texttt{>})$ be propositions on the state of an object `B` of class `C`.

Let $\mathtt{a}_i$ be message and let $p \in P$ be all free variables in the formulas with their scope.

We define five formula schemata for a class C with n methods:

$$\begin{aligned}
&\mathit{Persistence}(B) = (\nu X : \ldots : (\wedge p : p \in P : \\
&\quad \text{“<B:C>”} \implies [-](\text{“<B:C>”} \wedge X))) \\
&\mathit{State}(B) = (\nu X : \ldots : (\wedge p : p \in P : \\
&\quad \mathit{SI}(\text{“<B:C|atts>”}) \implies [-](\mathit{SI}(\text{“<B:C|atts'>”}) \wedge X))) \\
&\mathit{Synchronization}(B) = (\nu X : \ldots : (\wedge p : p \in P : (\wedge i : 1 \leq i \leq n : \\
&\quad \text{“<B:C|atts>”} \wedge \mathit{SI}(\text{<B:C|atts>}) \wedge \psi_i(\text{<B:C|atts>}) \\
&\qquad \implies \langle m_i \rangle X))) \\
&\mathit{StateChange}(B) = (\nu X : \ldots : (\wedge p : p \in P : (\wedge i : 1 \leq i \leq n : \\
&\quad \text{“<B:C|atts>”} \wedge \mathit{SI}(\text{<B:C|atts>}) \wedge \psi_i(\text{<B:C|atts>}) \\
&\qquad \implies [m_i](\phi_i(\text{<B:C|atts}_i\text{'>}) \wedge X)))) \\
&\mathit{AnswerMessages}(B) = (\nu X : \ldots : (\wedge p : p \in P : (\wedge i : 1 \leq i \leq n : \\
&\quad \text{“<B:C|atts>”} \wedge \mathit{SI}(\text{<B:C|atts>}) \wedge \psi_i(\text{<B:C|atts>}) \\
&\qquad \implies [m_i](\text{“}\mathtt{a}_i\text{”} \wedge X))))
\end{aligned}$$

A rather basic safety-property of a bounded buffer is persistence: if a bounded buffer is part of a configuration it is also part of all successor states:

$$\begin{aligned}
&\mathit{Persistence}(B) = \\
&(\nu X :: (\wedge \mathtt{I}, \mathtt{O}, \mathtt{M}, \mathtt{L} : \mathtt{I}, \mathtt{O}, \mathtt{M} \in \mathtt{Nat}, \mathtt{L} \in \mathtt{OIdList} : \\
&\qquad \text{“<B:BdBuffer>”} \implies [-]\text{“<B:BdBuffer>”} \wedge X))
\end{aligned}$$

The formula *State* specifies an invariant for the internal state of an object. It has to hold for all objects of a class in all states, provided it holds once, and it ensures consistency of the internal state. We do not care for bounded buffers which are in an inconsistent state. At this level we make the design decision that the state of a bounded buffer is implemented by four attributes, namely `in`, `out`, `max`, and `cont`.

$$\begin{aligned}
&\mathit{State}(B) = \\
&(\nu X :: (\wedge \mathtt{I}, \mathtt{O}, \mathtt{M}, \mathtt{L}, \mathtt{I'}, \mathtt{O'}, \mathtt{M'}, \mathtt{L'} : \\
&\qquad \mathtt{I}, \mathtt{O}, \mathtt{M}, \mathtt{I'}, \mathtt{O'}, \mathtt{M'}, \in \mathtt{Nat}, \mathtt{L}, \mathtt{L'} \in \mathtt{OIdList} : \\
&\quad \text{“<B:BdBuffer|in:I,out:O,max:M,cont:L>”} \\
&\quad \wedge \mathtt{length(L)} = \mathtt{I} - \mathtt{O} \wedge 0 \leq \mathtt{I} - \mathtt{O} \leq \mathtt{M} \\
&\qquad \implies [-]\text{“<B:BdBuffer|in:I',out:O',max:M',cont:L'>”} \\
&\qquad\quad \wedge \mathtt{length(L')} = \mathtt{I'} - \mathtt{O'} \wedge 0 \leq \mathtt{I'} - \mathtt{O'} \leq \mathtt{M'} \wedge X))
\end{aligned}$$

We refer to the formula $\texttt{length(L)} = \texttt{I} - \texttt{O} \wedge \texttt{O} \leq \texttt{I} - \texttt{O} \leq \texttt{M}$ as *SI*, (state invariant) of a bounded buffer.

In the formula *Synchronization* we specify the so-called synchronization code. The synchronization code determines when an objects accepts which message. This synchronization code depends in our approach only on the state of an object.

$$
\begin{aligned}
&\mathit{Synchronization}(B) = \\
&(\nu X :: (\wedge \texttt{I}, \texttt{O}, \texttt{M}, \texttt{L}, \texttt{E}, \texttt{E'} : \texttt{I}, \texttt{O}, \texttt{M} \in \texttt{Nat}, \texttt{L} \in \texttt{OIdList}, \texttt{E}, \texttt{E'} \in \texttt{OId} : \\
&\quad (\ \text{“}\texttt{<B:BdBuffer|in:I,out:O,max:M,cont:L>}\text{”} \\
&\quad \wedge\ \mathit{SI}(\texttt{<B:BdBuffer>}) \wedge \texttt{I} - \texttt{O} < \texttt{M} \\
&\quad \Longrightarrow \langle \texttt{(to B: put E)} \rangle X)) \\
&\quad \wedge (\ \text{“}\texttt{<B:BdBuffer|in:I,out:O,max:M,cont:L E'>}\text{”} \\
&\quad \wedge\ \mathit{SI}(\texttt{<B:BdBuffer>}) \wedge \texttt{I} - \texttt{O} > 0 \\
&\quad \Longrightarrow \langle \texttt{(to B: get)} \rangle X)))
\end{aligned}
$$

The two formulas *StateChange* and *Synchronization* specify the internal behavior of an object. From the synchronization code we obtain not only when a method may be invoked but also under which preconditions this method and the functions on data types must operate correctly on the state of the object. In the formula *StateChange*, we specify how methods change the state of an object. When a message is accepted it *always* changes the state of an object in the same way:

$$
\begin{aligned}
&\mathit{StateChange}(B) = \\
&(\nu X :: (\wedge \texttt{I}, \texttt{O}, \texttt{M}, \texttt{L}, \texttt{E}, : \texttt{I}, \texttt{O}, \texttt{M} \in \texttt{Nat}, \texttt{L} \in \texttt{OIdList}, \texttt{E} \in \texttt{OId} : \\
&\quad (\ \text{“}\texttt{<B:BdBuffer|in:I,out:O,max:M,cont:L>}\text{”} \\
&\quad \wedge\ \mathit{SI}(\texttt{<B:BdBuffer>}) \wedge \texttt{I} - \texttt{O} < \texttt{M} \\
&\quad \Longrightarrow [\texttt{(to B: put E)}] \\
&\qquad (\text{“}\texttt{<B:BdBuffer|in:I+1,out:O,max:M,cont:E L >}\text{”} \wedge X) \\
&\quad \wedge (\ \text{“}\texttt{<B:BdBuffer|in:I,out:O,max:M,cont:L E'>}\text{”} \\
&\quad \wedge\ \mathit{SI}(\texttt{<B:BdBuffer>}) \wedge \texttt{I} - \texttt{O} > 0 \\
&\quad \Longrightarrow [\texttt{(to B: get)}] \\
&\qquad (\text{“}\texttt{<B:BdBuffer|in:I,out:O+1,max:M,cont:L>}\text{”} \wedge X)))
\end{aligned}
$$

After a put action, the value of attribute in is incremented and the element that is parameter to the message is added to the contents. After a get message, the value of out is incremented and the element which is parameter in the message added to the value of attribute cont.

Messages not only change the internal state of the objects, they also trigger

answer messages to be created as part of the global state:

$$
\begin{aligned}
&\mathit{AnswerMessages}(B) = \\
&(\nu X :: (\wedge \mathtt{I}, \mathtt{O}, \mathtt{M}, \mathtt{E}, \mathtt{E'}, \mathtt{L} : \mathtt{I}, \mathtt{O}, \mathtt{M} \in \mathtt{Nat}, \mathtt{E}, \mathtt{E'} \in \mathtt{OId}, \mathtt{L} \in \mathtt{OIdList} : \\
&\quad (\ \text{“}\mathtt{<B:BdBuffer|in:I,out:O,max:M,cont:L>}\text{”} \\
&\quad \wedge\ \mathit{SI}(\mathtt{<B:BdBuffer>}) \wedge \mathtt{I} - \mathtt{O} < \mathtt{M} \\
&\quad \Longrightarrow\ [\mathtt{(to\ B:\ put\ E)}]X) \\
&\quad \wedge (\ \text{“}\mathtt{<B:BdBuffer|in:I,out:O,max:M,cont:L\ E'>}\text{”} \\
&\quad\quad \wedge\ \mathit{SI}(\mathtt{<B:BdBuffer>}) \wedge \mathtt{I} - \mathtt{O} > 0 \\
&\quad\quad \Longrightarrow\ [\mathtt{(to\ B:\ get)}] \\
&\quad\quad\quad (\text{“}\mathtt{(to\ U:\ answer\ to\ get\ is\ E)}\text{”} \wedge X)))
\end{aligned}
$$

After the message `(to P: get)`, message “`(answer to get is E)`” is part of the global state of the system. Again, we specify that after a method is invoked an answer message is *always* part of the (global) state.

The specification at the intermediate level is given by

$$
\begin{aligned}
\Phi_M = \{\, &\mathit{State}(B), \mathit{Persistence}(B), \\
&\mathit{StateChange}(B), \mathit{AnswerMessages}(B), \mathit{Synchronization}(B)\}
\end{aligned}
$$

After giving those five formulas as a specification of a class, we would like to give a brief motivation why we use both diamond and box operators for modeling the different aspects of an object. The use of the diamond operator is quite easy to motivate: we are interested in which state transitions are possible for an object, which transitions an object may perform. This is the kind of property expressible by the diamond operator.

The use of the box operator needs more motivation and we give two reasons for preferring the box to the diamond operator for specifying the internal properties and behavior of objects. The first motivation is that, typically, even in object-oriented concurrent languages, an object is sequential and deterministic. Thus a property $\langle \mathtt{get} \rangle \langle \mathtt{answer\ to\ get\ is\ E} \rangle \phi$ would not reflect the situation that after a `get` action there is always an `answer` message possible for the overall system.

The second motivation for the use of the box operators lies in the properties of the overall system we are interested in. One very important property is absence of deadlocks specified by

$$\mathit{Deadlockfree} = (\nu X : \ldots : \langle - \rangle \mathtt{tt} \wedge [-]X)$$

Let us explain this formula: in every state, a transition (with arbitrary label) is possible and after every transition the property *Deadlockfree* is satisfied.

Our schemata and formulas specify the behavior of a single class but they do not specify the behavior of the global system. From the global point of view a message must be part of the state to make the local transition of an object possible, provided the precondition specified in *Synchronization* holds

as well. Assume we have in our specification an object that consumes always all **answer** messages, i.e., if a message is part of the global state then there is always a transition with this label possible.

If we use only diamond properties to specify the behavior of an object we would obtain the property

$$\langle\texttt{(to B get)}\rangle\texttt{tt} \wedge \langle\texttt{(to B get)}\rangle\langle\texttt{(answer to get is E)}\rangle\texttt{tt}$$

With the box operator we obtain

$$\langle\texttt{(to B get)}\rangle\texttt{tt} \wedge [\texttt{(to B get)}]\langle\texttt{(answer to get is E)}\rangle\texttt{tt}$$

This formula, which uses the box operator, is stronger and models the absence of a deadlock in a situation when a user waits for the bounded buffer.

Deadlocks are typically caused by the composition of a system as a (large) collection of objects belonging to different classes and deadlocks inside objects are not really an issue in specification. The box property specifying the internal behavior gives us a property which is important when composing a large system.

4.3 The concrete level – Maude

Maude's transition rules and rewriting calculus provide only the possibility to express sometime-properties on single actions, but not always-properties and not properties on sequences of actions. Each transition rule specifies the reaction (or one way to react to a message) of an object to a message. Thus we have to refine the intermediate specification which focuses on certain aspects of the behavior of a class to a specification which focuses on the local and global reaction to a message.

There is one more, severe difference between the concrete level of Maude and the object-oriented μ-specification: The rules of Maude describe the global transition system, and the formulas in structured μ-calculus properties of a single class.

Lemma 4.2 $Sp_M \rightsquigarrow$ *BDBUFFER*.

Proof. $Sp_M \rightsquigarrow$BDBUFFER iff

$$\begin{aligned} \mathrm{I}(\texttt{BDBUFFER}) \models\ & \mathit{Persistence}(B) \wedge \mathit{State}(B) \\ & \wedge\ \mathit{Synchronization}(B) \\ & \wedge\ \mathit{StateChange}(B) \wedge \mathit{AnswerMessages}(B) \end{aligned}$$

We proceed as follows:

(i) We strengthen the formulas *Synchronization*, *StateChange*, *AnswerMessages* such that each precondition requires additionally the presence of the appropriate message in the global state. (The formulas are not given explicitly and their names are primed.)

(ii) We show that we may omit the fixpoint operator in the formulas in our particular setting.

(iii) We show that a formula containing a diamond operator $\langle m \rangle$ implies a formula containing a box operator $[m]$, provided there is always just one successor state reachable via a transition with label m.

(iv) We compose the strengthened formulas to a formula *Rule* such that *Rule* implies all three strengthened formulas.

(v) We prove that $I(\texttt{BDBUFFER}) \models \mathit{Rule}$.

(vi) We prove that $I(\texttt{BDBUFFER}) \models \mathit{Persistence}(B) \wedge \mathit{State}(B)$. (The proof is not given)

Step i: Is not given here.

Step ii: Let (A, R) be $I(Sp)$ of some specification Sp and ϕ a μ-formula. Then $(A, R) \models \phi$ iff for all $C \in A_{\mathrm{Cf}}$ holds $(A, R), C \models \phi$

We prove: $(A, R), C \models \mathit{pre} \implies \langle L \rangle \mathit{post}$ for all $C \in A_{\mathrm{Cf}}$

implies $(A, R) \models (\nu X :: \mathit{pre} \implies \langle L \rangle (\mathit{post} \wedge X))$

$(A, R) \models (\nu X :: \mathit{pre} \implies \langle L \rangle (\mathit{post} \wedge X))$
$\Leftrightarrow$ { Definition of $\models$ }
$(A, R), C \models (\nu X :: \mathit{pre} \implies \langle L \rangle (\mathit{post} \wedge X))$
for all $C \in A_{\mathrm{Cf}}$
$\Leftrightarrow$ { Definition of $\models$ }
$C \in (\cup C' : C' \subseteq A_{\mathrm{Cf}}, \mid \mathit{pre} \implies \langle L \rangle (\mathit{post} \wedge X) \mid_{(A,R),X:=C'} \supseteq C' : C')$
for all $C \in A_{\mathrm{Cf}}$
$\Leftrightarrow$ { $S \subseteq S'$ and $C \in S$ for all $C \in S' \Leftrightarrow S = S'$ }
$A_{\mathrm{Cf}} = (\cup C' : C' \subseteq A_{\mathrm{Cf}}, \mid \mathit{pre} \implies \langle L \rangle (\mathit{post} \wedge X) \mid_{(A,R),X:=C'} \supseteq C' : C')$
$\Leftarrow$ { $C' = A_{\mathrm{Cf}}$ }
$A_{\mathrm{Cf}} = \mid \mathit{pre} \implies \langle L \rangle (\mathit{post} \wedge X) \mid_{(A,R),X:=A_{\mathrm{Cf}}}$
$\Leftrightarrow$ { }
$A_{\mathrm{Cf}} = \mid \mathit{pre} \implies \langle L \rangle \mathit{post} \mid_{(A,R),X:=A_{\mathrm{Cf}}}$

Thus, we may omit the fixpoint operator in the three formulas *Synchronization*, *AnswerMessages* and *StateChange*.

Step iii: In our specification BDBUFFER each method is implemented in only one rule, and thus $(A, R), C \models \langle l \rangle \phi \implies (A, R), C \models [l] \phi$ if l is a singleton set of labels.

Step iv: We define a new formula schema :

$$\begin{aligned} \mathit{Rule} = (\wedge\, B, p : B \in \texttt{OId}, p \in P : (\wedge i : 1 \leq i \leq n : \\ \text{“}\texttt{<B:C|atts>}\text{”} \wedge \mathit{SI}(\texttt{<B:C|atts>}) \wedge \psi_i(\texttt{<B:C|atts>}) \\ \implies \langle m_i \rangle (\phi_i(\texttt{<B:C|atts}_i\texttt{'>}) \wedge \text{“}\texttt{a}_i\text{”}))) \end{aligned}$$

such that

$$\begin{aligned} \mathit{Rule} \implies & \mathit{Synchronization}'(B) \wedge \mathit{StateChange}'(B) \\ & \wedge \mathit{AnswerMessage}'(B) \end{aligned}$$

Step v:
To prove: for all $C \in A_{\mathrm{Cf}}$ holds
$(A, R), C \models$

$(\wedge \mathtt{I}, \mathtt{O}, \mathtt{M}, \mathtt{L}, \mathtt{E}, \mathtt{E'} : \mathtt{I}, \mathtt{O}, \mathtt{M} \in \mathtt{Nat}, \mathtt{L} \in \mathtt{OIdList}, \mathtt{E}, \mathtt{E'} \in \mathtt{OId} :$

$((\text{“}\mathtt{(to\ B:\ put\ E)}\text{”} \wedge \text{“}\mathtt{<B:BdBuffer|in:I,out:O,max:M,cont:L>}\text{”}$

$\wedge SI(\mathtt{<B:BdBuffer>}) \wedge \mathtt{I} - \mathtt{O} < \mathtt{M})$

$\Longrightarrow \langle \mathtt{(to\ B:\ put\ E)} \rangle$

$\text{“}\mathtt{<B:BdBuffer|in:I+1,out:O,max:M,cont:E\ L>}\text{”})$

$\wedge$ $((\text{“}\mathtt{(to\ B:\ get)}\text{”} \wedge \text{“}\mathtt{<B:BdBuffer|in:I,out:O,max:M,cont:L\ E'>}\text{”}$

$\wedge SI(\mathtt{<B:BdBuffer>}) \wedge \mathtt{I} - \mathtt{O} > 0)$

$\Longrightarrow \langle \mathtt{(to\ B:\ get)} \rangle (\text{“}\mathtt{(answer\ to\ get\ is\ E')}\text{”}$

$\wedge \text{“}\mathtt{<B:BdBuffer|in:I,out:O+1,max:M,cont:L\ >}\text{”}))$

Proof by structural induction on the size of configurations.

Case 1 : $C = \mathtt{eps}$ (the empty configuration)

$(A, R), C \models (\wedge \ldots : \ldots : \mathtt{ff} \Longrightarrow \langle \mathtt{(to\ B:\ put\ E)} \rangle \ldots$

$\wedge \mathtt{ff} \Longrightarrow \langle \mathtt{(to\ B:\ get)} \rangle \ldots)$

Case 2 : $C \neq \mathtt{eps}$, no state transition possible.

No state transition possible if there exists no σ, such that σ applied to the left-hand side of rule `[get]` or `[put]` is part of C.

And, thus, for all valuations v

$(A, R), C, v \not\models \text{“}\mathtt{<B:BdBuffer|in:I,out:O,max:M,cont:L>}\text{”}$

$\wedge SI(\mathtt{<B:BdBuffer>}) \wedge (\text{“}\mathtt{(to\ B\ get)}\text{”} \vee \text{“}\mathtt{(to\ B\ put\ E)}\text{”})$

and, thus, $(A, R), C \models \mathit{Rule}$

Case 3.1 : $C = \mathtt{<b:BdBuffer|in:i,out:o,max:m,cont:l>\ (to\ b\ put\ e)}$ such that the transition rule with label `[put]` is applicable (small letters denote values)

Then $\sigma = [\mathtt{B} \to \mathtt{b}, \mathtt{I} \to \mathtt{i}, \mathtt{O} \to \mathtt{o}, \mathtt{M} \to \mathtt{m}, \mathtt{L} \to \mathtt{l}, \mathtt{e} \to \mathtt{E}]$ is the only substitution such that rule (Inst) of the calculus is applicable.

Let $D = \sigma(\mathtt{<B:BdBuffer|in:I,out:O+1,max:M,cont:L>}$

$\sigma(\mathtt{(answer\ to\ get\ is\ E)}))$

Thus, $(C, D) \in R$ and

$(A, R), D \models \text{“}\mathtt{(answer\ to\ get\ is\ e)}\text{”}$

$\wedge \text{“}\mathtt{<b:BdBuffer|in:i,out:o+1,max:m,cont:l>}\text{”}$

and thus $(A, R), C \models \mathit{Rule}$

Case 3.2 : $C = \mathtt{<b:BdBuffer|in:i,out:o,max:m,cont:l>\ (to\ b\ get)}$

Analogously to Case 3.1.

Case 4 : $C = C_1\ C_2$

For all propositional variables ϕ and for $\langle L \rangle \phi$ holds:

$\phi(C_i) \Longrightarrow \phi(C)$ and $(A, R), C_i \models \langle L \rangle \phi \Longrightarrow (A, R), C \models \langle L \rangle \phi$

And by application of rule (Emb): $C_i \xrightarrow{m} D_i \Longrightarrow C \xrightarrow{m} D_i\ C_j (i \neq j)$.

Thus we consider only a transition which is generated by an application

of (Inst) and (Emb) which not applicable by (Inst) to C_1 or C_2, This proof follows cases 3.1 or 3.2.
Step vi: Proof by induction on the size of configurations. □

5 Related Work

The main issues in our work are (1) the specification of objects (2) the expressiveness of specification languages for concurrent systems and (3) refinement of specifications.

In the specification of objects and, in particular, of the bounded buffer we combine, in all three levels of specification, algebraic specification [32] and specification languages for concurrent systems. The SMoLCS approach [2,3] combines also algebraic specification and specification of transition systems. The SMoLCS approach is designed for modular specification of the semantics of programming languages. Particular to the approach is the way the transition systems that model the behavior of a program are specified. Since algebras and transition systems are also the semantic foundations of our specifications, the SMoLCS approach could be used to specify a semantics for our μ and Maude specifications.

When comparing our object model and our two specification languages, Maude and the μ-calculus, to other object-oriented concurrent approaches, we notice that most other approaches like Troll [13], $\pi o \beta \lambda$ [18] use synchronous communication. In this respect, our μ and Maude specifications are related more closely to actor languages [1,11]. Troll and $\pi o \beta \lambda$ specify the implementation of methods inside objects and focus thus in specification on the intra-object view with properties of classes, while our approach abstracts from the implementation of methods and provides together with [20] an intra- and inter-object view.

The use of greatest fixpoints is inspired by the coalgebraic specifications of classes in [16,17,31]. While an algebraic specification specifies the properties of a class [6] the coalgebraic specification gives the observable behavior and properties. Thus the use of greatest fixpoint and the coalgebraic specification style reflect the principle of encapsulation, the basic concept of object orientation, more than the algebraic approach.

Refinement relations have been studied in various versions for algebraic specifications [4,15,32]. The initial and the loose approach to refinement are compared in [30]. The initial approach to refinement of Maude specifications is used in [24,33]. Refinement of object-oriented languages is studied also for Troll [8,9] and $\pi o \beta \lambda$[18]. These refinement approaches are mainly concerned with action refinement and refinement of communication between objects. Their correctness criterium of the refinement is the state of the global system, but not, like in our approach the behavior of a class. While all these approaches as well as our approach remain in the formal world of specification is in [33] an abstract informal specification in the object-oriented analysis method of Jacobsen refined to a Maude specification and from the Maude specification to a Java program.

6 Conclusions

Both Maude and the μ-calculus are specification languages that can be used for the specification of concurrent systems. While Maude is designed to make it executable, the advantage of the μ-calculus is its expressiveness. But, when we combine these two specification mechanisms, as in the intermediate level of our specification, we get a very expressive language, appropriate for specification. At this level, it should be relatively easy to find operators for various property-preserving kinds of reuse as, e.g., inheritance and, thus, *modal-μ-Maude* could be a specification language on its own.

Acknowledgement

We would like to thank the anonymous referees for their helpful comments. Many fruitful discussions with Christian Lengauer and Martin Wirsing helped to improve this work.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Artificial Intelligence. MIT Press, 1986.

[2] E. Astesiano, G.F. Mascari, G. Reggio, and M. Wirsing. On the parameterized algebraic specification of concurrent systems. In H. Ehrig, C. Floyd, M. Nivat, and M. Thatcher, editors, *Mathematical Foundations of Software Development*, Lecture Notes in Computer Science 185, pages 342–358. Springer-Verlag, 1985.

[3] E. Astesiano and M. Wirsing. Bisimulation in algebraic specifications. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, Algebraic Techniques, Vol. 1, pages 1–32. Academic Press, London, 1989.

[4] M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 2-3(25):146–186, 1995.

[5] J.C. Bradfield. *Verifying Temporal Properties of Systems.* Birkhäuser, 1992.

[6] R. Breu. *Algebraic Specification Techniques in Object-Oriented Programming Environments.* Lecture Notes in Computer Science 562. Springer-Verlag, 1991.

[7] G. Bruns. A practical technique for process abstraction. In E. Best, editor, *4th Int. Conf. on Concurrency Theory (CONCUR'93)*, Lecture Notes in Computer Science 715, pages 37–49. Springer-Verlag, 1993.

[8] G. Denker. A semantic characterisation of correct refinement of object specifications based on database schedules. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence.* Kluwer, 1996. To appear.

[9] G. Denker and H.-D. Ehrich. Action Reification In Object Oriented Specifications. In *Proc. ISCORE Workshop Amsterdam, Sep. 94.* World Scientific, 1995.

[10] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[11] S. Frølund. Inheritance of synchronisation constraints in concurrent object-oriented programming languages. In O. Lehrmann Madsen, editor, *European Conf. on Object-Oriented Programming (ECOOP'92)*, Lecture Notes in Computer Science 615, pages 185–196. Springer-Verlag, 1992.

[12] J.A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[13] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. Revised version of the modelling language TROLL (Version 2.0). Technical Report Informatik-Bericht 94–03, TU Braunschweig, 1994.

[14] A.E. Haxthausen and F. Nickl. Pushouts of order-sorted algebraic specifications. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST 96)*, Lecture Notes in Computer Science 1101. Springer-Verlag, 1996.

[15] M. Hofmann and D. Sannella. On behavioral abstraction and behavioral satisfaction in higher-order logic. In *Proc. Theory and Practice of Software Development (TAPSOFT'95)*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.

[16] B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on object-oriented programming (ECOOP)*, Lecture Notes in Computer Science 1098, pages 210–231. Springer-Verlag, 1996.

[17] B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence.* Kluwer, 1996. To appear.

[18] C.B. Jones. Constraining Interference in an Object-Based Design Method. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Theory and Practice of Software Development (TAPSOFT'93)*, Lecture Notes in Computer Science 668, pages 136–150. Springer-Verlag, 1993.

[19] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[20] U. Lechner and C. Lengauer. Modal–μ–Maude — properties and specification of concurrent objects. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence.* Kluwer, 1996. To appear.

[21] U. Lechner, C. Lengauer, F. Nickl, and M. Wirsing. (Objects + Concurrency) & Reusability – A Proposal to Circumvent the Inheritance Anomaly. In *ECOOP'96*, Lecture Notes in Computer Science 1098, pages 232–248. Springer-Verlag, 1996.

[22] U. Lechner, C. Lengauer, and M. Wirsing. An Object-Oriented Airport: Specification and Refinement in Maude. In E. Astesiano, G. Reggio, and

A. Tarlecki, editors, *10th Workshop on Specification of Abstract Data Types Joint with the 5th COMPASS Workshop, Selected papers*, Lecture Notes in Computer Science 906, pages 351–367. Springer-Verlag, 1995.

[23] C. Loiseaux, A. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstraction for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–45, 1995.

[24] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. Technical Report SRI-CSL-92-08, SRI International, July 1992.

[25] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[26] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O. Nierstrasz, editor, *ECOOP '93 – Object-Oriented Programming*, Lecture Notes in Computer Science 707, pages 220–246. Springer-Verlag, 1993.

[27] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Concur 96*, Lecture Notes in Computer Science. Springer-Verlag, 1996. To appear.

[28] J. Meseguer and T. Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, Lecture Notes in Computer Science 574, pages 253–293. Springer-Verlag, 1992.

[29] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 25:267–310, 1993.

[30] F. Orejas, M. Navarro, and A. Sanchez. Implementation and behavioural equivalence: A survey. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specifications*, Lecture Notes in Computer Science 655, pages 93–125. Springer-Verlag, 1993.

[31] H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. Comp. Sci.*, 5:129–152, 1995.

[32] M. Wirsing. Algebraic specification. In J.V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier (North-Holland), 1990.

[33] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. *ENTCS*, 4, 1996. In this volume.

A Specification of OIDLIST

```
obj OIDLIST is
  protecting OID .
  sort OIdList .

  op eps      : -> OIdList  .
  op _ _      : OId OIdList -> OIdList . (* left append *)
  op _ _      : OIdList OId -> OIdList . (* right append *)
  op length   : OIdList -> Nat .

  var E E1 E2 : OId .
  var L : OIdList .

  eq E2 eps = eps E2 .
  eq E1 (L E2)  = (E1 L) E2 .

  eq length (eps) = 0 .
  eq length (E L) = succ(length(L)).
endfm
```

Note that we do not use subsorting to implement lists. The reason for this is that in the presence of subsorting, the union operation, in general, does not preserve the coherence of signatures [14,21].

Electronic Notes in Theoretical Computer Science 4 (1996)

A Maude specification of an object-oriented database model for telecommunication networks

Isabel Pita [1] and Narciso Martí-Oliet [2]

Departamento de Informática y Automática
Escuela Superior de Informática
Universidad Complutense de Madrid, Spain
{ipandreu,narciso}@eucmos.sim.ucm.es

Abstract

This paper presents an object-oriented database model for broadband telecommunication networks, which can be used both for network management and for network planning purposes. The object-oriented data model has been developed using the parallel object-oriented specification language Maude [8,11], which allows us to define not only structural aspects of the database, but also procedural aspects. Several modeling approaches are compared, emphasizing the definition of the object relationships and some of the procedural aspects of the model.

1 Introduction

Maude is a specification language based on rewriting logic [7], which integrates equational and object-oriented programming in a satisfactory way. Its logical basis facilitates a clear definition of the object-oriented semantics and makes it a good choice for the formal specification of object-oriented systems.

Rewriting logic was first proposed by Meseguer as a unifying framework for concurrency in 1990 [6]. Since then much work has been done on the use of rewriting logic as a logical and semantic framework [5,10], and on the development of the Maude language [8,12], but few studies have been done on the application of rewriting logic and Maude to the specification of real systems, except for the airport specification described in [13,3].

The idea of using Maude to specify object-oriented databases was introduced by Meseguer and Qian in [11]. The key advantages of this approach are the fact that dynamic aspects can be integrated in the database model, the integration of algebraic data types within the framework, and the integration

[1] This paper summarizes work done by the first author in her master thesis.
[2] Partially supported by CICYT, TIC 95-0433-C03-01.

© *1996 Elsevier Science B. V.*

of query, update and programming aspects within a single language, which allows us to formulate queries and updates to the database in Maude.

We develop a Maude specification of a database model for broadband telecommunication networks showing the power of the language to specify this kind of systems, and in particular the well known inheritance relationship and other object relationships, like containment or symmetric relationships ("member-of," "client-server," ...). The latter, although not inherent to object-oriented programming, facilitate and clarify the specification of systems by making explicit the relationships between components in the specification instead of hiding them in the code during the implementation process. The selected application of modeling broadband telecommunication networks is a good choice to illustrate containment and symmetric relationships because they appear in a natural way between the objects of different layers of the network model and between objects of the same layer.

Because the object-oriented classes defined in the model are taken from the standard classes specified by the International Telecommunications Union (ITU) for the interconnection of telecommunication systems [1], the resulting model is very general and can be used for the integration of many different applications. Developed as the information model of the object-oriented management protocols used by the systems, it has become the best choice for the database model of the network management centers, to avoid information model translations. For the same reason it is also very appropriate for applications that can interact with this database, such as network planning or the simulation of the network behaviour under different situations.

This paper is organized as follows. First we present some basic notions of Maude that will be used in the specification process. Then we introduce the system that will be specified by defining the model of the network and describing the object-oriented classes defined in the system, and the relationships between them. We also specify some possible queries to the database that are realized by methods implemented in the classes. Finally the implementation of different types of object relationships in Maude is discussed.

2 The Maude language

We outline here some basic notions of the object-oriented part of Maude that are used in the application. For more information on the language see [8,12].

Systems in Maude are built out of basic elements called modules. *Functional modules* are used for the definition of algebraic data types and *object-oriented modules* for the definition of object-oriented classes. An object-oriented module consist of an import list of modules, class declarations, message declarations, and rewrite rules, for which the types (called sorts) of the variables appearing in terms are also declared. A class declaration includes the class identifier, attribute identifiers, and the type of each attribute which can be an algebraic data type defined in a functional module, or an entire configuration defined in another object-oriented module.

An object is represented as a term `< O : C | a1: v1, ..., an: vn>`,

where `O` is the object's name belonging to a set `OId` of object identifiers, `C` is the class identifier, the `ai`'s are the names of the object's attributes, and the `vi`'s are their corresponding values, which typically are required to be in a sort appropriate for their corresponding attribute.

Rewrite rules represent the real code of the module, that is, the implementation of the method associated to a message received by an object. In general a rewrite rule has the form

$$\begin{aligned} \text{rl}\ & M_1 \ldots M_n < O_1 : C_1 \mid atts_1 > \ldots < O_m : C_m \mid atts_m > \\ \longrightarrow\ & < O_{i1} : C'_{i1} \mid atts'_{i1} > \ldots < O_{ik} : C'_{ik} \mid atts'_{ik} > \\ & < Q_1 : D_1 \mid atts''_1 > \ldots < Q_p : D_p \mid atts''_p > \\ & M'_1 \ldots M'_q \\ & if\ C \end{aligned}$$

where $k, p, q \geq 0$, the M_s are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is a rule condition. The result of applying a rewrite rule is that

- the messages $M_1, \ldots, M_n$ disappear;
- the state and possibly the class of the objects $O_{i1}, \ldots, O_{ik}$ may change;
- all the other objects O_j vanish;
- new objects $Q_1, \ldots, Q_p$ are created;
- new messages $M'_1, \ldots, M'_q$ are sent.

By convention, the only object attributes $atts_1 \ldots atts_m$ made explicit in a rule are those relevant for that rule. In particular, the attributes mentioned only on the lefthand side of the rule are preserved unchanged, the original values of attributes mentioned only on the righthand side of the rule don't matter, and all attributes not explicitly mentioned are left unchanged.

With respect to the inheritance relationship, Maude distinguishes two kinds of inheritance: *class inheritance* and *module inheritance*. Class inheritance is directly supported by Maude's order-sorted structure. The effect of a subclass declaration is that the attributes, messages and rules of all the superclasses contribute to the structure and behaviour of the objects in the subclass, and cannot be modified in the subclass; in addition, the subclass can have new attributes and messages. On the other hand, module inheritance is used for code reuse, allowing the modification of the original code in several ways.

3 The network model

We use Maude to specify the database model of a broadband telecommunication network. Due to their complexity, this kind of networks are modelled in layers that separate different communication aspects and simplify the global treatment. The selected model is based on the layered structure of broadband networks, which divides the network into three logical layers: *physical* layer, *virtual path* (VP) layer, and *virtual channel* (VC) layer, each one related to

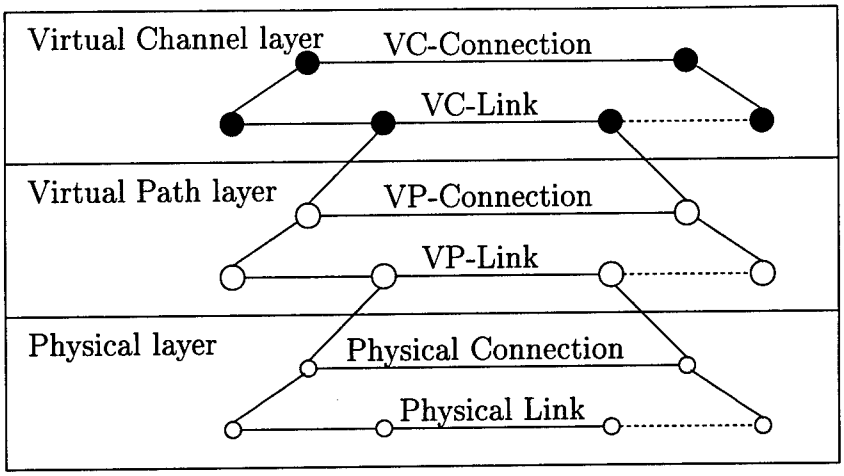


Fig. 1. Network structure

different network functions (Figure 1).

3.1 Network objects

The basic objects of the model are *nodes*, *links* and *connections*, all of which appear in each of the three network layers. Nodes represent the network points where the communication signal is treated. Two kinds of nodes are distinguished:

- Transmission nodes, which have physical and transmission functions and are defined in the physical layer.
- Switching nodes, which have switching and crossconnection functions and are defined in the upper layers.

Nodes of the upper layers are placed on lower node locations, making the node locations defined on a layer a subset of the locations defined on the lower layer.

Links are defined between nodes of the same layer as manageable entities for which the carried information can be accessed at its two end points. Although it is not necessary to define a link between each pair of nodes in a layer, a connected graph is required.

Connections are defined as a configurable sequence of links in the same layer. They are used to support the communication services between each pair of users, where each user is related to a unique node. There should be a connection defined between each pair of nodes in the network, as it is assumed that the users related to one node can communicate with users related to each of the other nodes. Upper layer links are supported by connections of the layer below, while physical links are supported by the transmission media.

In addition to the three basic objects used to define the topological network structure, there are other components of the network:

- Nodes are formed by pieces of equipment which represent the physical characteristics (number of ports in a termination unit, port capacity, ...) and the cost of the elements defined in a node. The equipment of a transmission

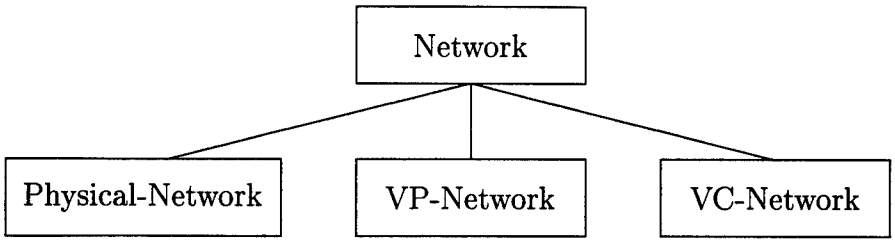


Fig. 2. Network inheritance relationship

node is called *transmission equipment*, whereas the equipment of a switching node is called *switching equipment*.

- The links of the physical layer are supported by the transmission media of the network. The *cable* component represents the characteristics and cost of this medium.
- The *services* represent the characteristics of the communication services (fax, telephone, mail, ...) provided by the network.
- The *locations* are the physical places where the nodes are located. As it was said before, locations of upper layers are always a subset of the lower layer locations.

A *network* is formed by a set of links together with the nodes that they join and the corresponding connections between the nodes. A different network is defined for each layer of the model: *physical network*, *VP network* and *VC network*. The links defined in each network are called respectively physical links, VP links, VC links, and analogously for nodes and connections.

3.2 Network objects relationships

Three types of relationships between network objects are defined in ITU recommendation X.720 [1]: inheritance relationship, containment relationship, and explicit group relationships [4].

The *inheritance* relationship captures the common properties of a set of classes by defining an order between them. Inheritance appears in the model between generic classes and the respective specialized classes for each layer. For example, Figure 2 shows a generic class Network that has three subclasses: Physical-Network, VP-Network, and VC-Network. Similar relationships are defined between the generic classes of nodes, links and connections, and the respective specialized classes for each layer.

The *containment* relationship, also called *object composition*, captures the semantics associated with the "is-part-of" relationship between objects. In its strongest sense, object composition implies that the part cannot exist without its owner, nor be shared with another owner, that is, composite objects must be destroyed when the owner object is destroyed and may only be created as part of the creation process of the owner.

In telecommunication networks it is a common practice to apply the containment relationship to the naming process, that is, to the construction of

the object identifier. OIds are the concatenation of the owner object OId with a unique contained object OId. The uniqueness of OIds for contained objects applies only inside the domain of the owner object. We will apply it to node equipment and its parts. It can also be used to model the relationship between the network and its elements (nodes, links, and connections) and between objects of different layers, but, because it imposes strong restrictions on the model, we have chosen explicit group relationships to model them.

Explicit group relationships are object relationships that do not involve any existence bonds or permanent associations, do not constrain creation and deletion operations, and allow multiple ownership. These relationships can be of different kinds: "client-server," "member-of," "back-up," .... The "client-server" relationship is used to represent the relation between different layers. The "member-of" relationship is defined naturally between links and connections of the same layer, and between the network and the set of nodes, links and connections that form it. "Back-up" relationships are defined between links of the same layer and between connections of the same layer.

4 The network specification

The network objects and their relationships constitute the conceptual schema of the database. The different network structures that can be defined are the database configurations. The database evolves by the requests (queries, modifications, deletions) that produce a chain of messages between the database objects until a new stable configuration corresponding to the request is reached.

4.1 The object-oriented classes

In general, each kind of network object described in Section 3.1 gives rise to a class in the object-oriented specification, and in turn each class is specified in a corresponding object-oriented module in Maude. In addition, we need several functional modules to specify algebraic data types such as lists, sets, tuples, etc.

The main object class of the model is the class `C-Network` which acts as the root class of the database, that is, all accesses to the database are realized through this class or through its derived classes. The class is specialized for each of the network layers in classes `C-Physical-Network`, `C-VP-Network` and `C-VC-Network`, as explained above.

Each network object in the class `C-Network` is characterized by all nodes, links and connections that belong to the network.

```
class C-Network | NodeSet : NodeOIdSet,
                  LinkSet : LinkOIdSet,
                  ConnectionSet : ConnectionOIdSet .
```

Because order does not matter on these groups of elements, the types of the attributes that define them are sets of object identifiers rather than lists of identifiers. Declaring sets of object identifiers is sufficient in the proposed model, because the relationship between these elements and the network is a

"member-of" relationship. If a containment relationship had been selected, it would have been better to represent the elements by sets of objects and represent them as subconfigurations [8,2], as discussed later in Section 5.2.

The messages associated to the class `C-Network` represent the queries and updates that can be done in the database, as shown in the following examples.

4.2 Query messages

Two query messages are presented: the first one obtains the value of an attribute that is directly defined in the database, and the second one calculates the required value from existing attribute values. In both cases, the query is sent to a network object, as the root of the database, which in turn sends messages to the appropiate network components to obtain the required values. Then it summarizes the information contained in the returning messages in order to forward a unique message to the external object that made the original query.

The message `LinkLoad?(O,N,L)` obtains the load of a selected link `L` in a network `N` and replies to an external object `O`. Two rules are used to implement this message:

```
rl LinkLoad?(O,N,L) < N : C-Network | LinkSet: L LS >
=> < N : C-Network > (L.Load reply to N and O) .

rl (to N and O,L.Load is LD)
   < N : C-Network | LinkSet: L LS >
=> < N : C-Network > (To O LinkLoad L is LD in N) .
```

where the used variables are declared by

```
var O : OId .                    var LS : LinkOIdSet .
var N : NetworkOId .             var LD : Nat .
var L : LinkOId .
```

The first one forwards the query to the appropriate link object. This rule only applies when the link `L` appears in the network set of links, given by the `LinkSet` attribute, as it is required on the lefthand side of the rule by declaring the value of the `LinkSet` attribute to be `L LS`, where `L` is the link of the message and `LS` is a set of link identifiers. The second rule receives the acknowledgement of the link object indicating the value `LD` of the `Load` attribute, and forwards the corresponding acknowledgement to the object that made the original query to the database. In this way the only database objects that interact with external objects are the `C-Network` objects.

The messages `(L.Load reply to N and O)` and `(to N and O,L.Load is LD)` obtain the value of the attributes directly defined in a class that are not hidden—in this case the value of the `Load` attribute of a link `L`—and send it to the network object `N`. The external object `O` is necessary in the message to identify the appropriate return address.

The message `NodeLinks?(O,N,NO)` obtains all the links in the network `N` that have a selected node `NO` as their origin or as their destination. The entire

database should be investigated. A message is broadcasted to all links in the network N, and the links ending in node NO are collected and forwarded to the external object O. The corresponding rules are

```
rl NodeLinks?(O,N,NO)
   < N : C-Network | NodeSet: NO NS, LinkSet: LS >
=> FindLink(O,N,NO,LS,null) .

rl FindLink(O,N,NO,L LS,LS1)
   < L : C-Link | Nodes: <<M1;M2>> >
=> < L : C-Link > FindLink(O,N,NO,LS,L LS1)
   if (M1 == NO) or (M2 == NO) .

rl FindLink(O,N,NO,L LS,LS1)
   < L : C-Link | Nodes: <<M1;M2>> >
=> < L : C-Link > FindLink(O,N,NO,LS,LS1)
   if (M1 =/= NO) and (M2 =/= NO) .

rl FindLink(O,N,NO,null,LS)
=> (To N and O NodeLinks NO are LS) .

rl (To N and O NodeLinks NO are LS)  < N : C-Network >
=> < N : C-Network > (To O in N Node NO Links LS) .
```

where the used variables are declared by:

```
var O : OId .                    var NS : NodeOIdSet .
var N : NetworkOId .             vars LS LS1 : LinkOIdSet .
vars NO M1 M2 : NodeOId .
```

The first and last rules are implemented in the object-oriented module defining the class C-Network. Two attributes of the C-Network class are used in the first rule: NodeSet, which declares the set of nodes that belong to the network and whose value for the network N includes the node identifier NO given in the message and a set of node identifiers NS; and LinkSet, which declares the set of links that belong to the network and is represented by a set of link identifiers LS.

The three rules implementing the message FindLink belong to the object-oriented module defining the class C-Link. Once the network object N broadcasts the query to all the associated links, the rules for FindLink collect the links that have the given node as one of its end points by adding them one at a time to the set of link identifiers defined in the last parameter of the FindLink(O,N,NO,LS,LS1) message, and send a new message to the remaining links in the set LS until there are no more links in this set. When all the network links have been treated, a message (To N and O NodeLinks NO are LS) with the set LS of all links that fullfil the condition is sent to the network N, which in turn replies to the external object O.

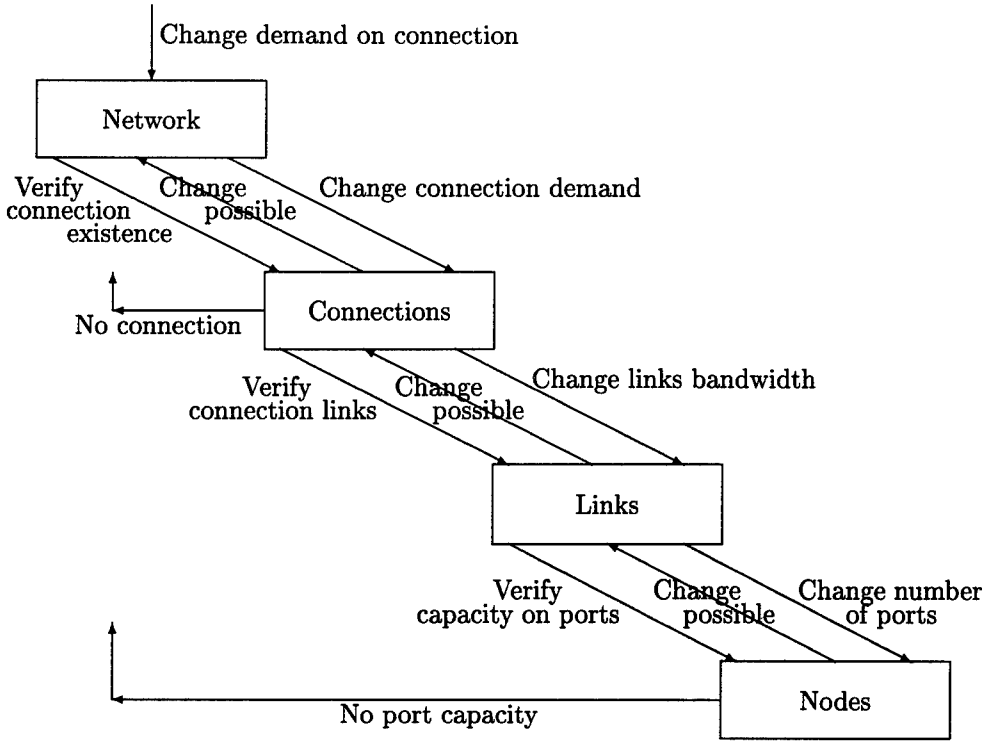


Fig. 3. Modification process

4.3 Modification messages

The message `ChDemand(O,N,NO1,NO2,<<S;D>>)` changes the communication demand of the connection between nodes `NO1` and `NO2` in the network `N`, by adding the demand `D` of service `S` (expressed as a tuple `<<S;D>>`) to the existing connection demand. If the demand is indeed changed, the message `(To O ChDemandOf C in R)` is sent to the external object; otherwise, a message indicating the reason why the change has not been done is sent. Two messages are used: `(To O NoConnectionBetweenNodes NO1 and NO2 in N)` for the case in which there is no connection defined between the given nodes and `(To O NoPortCapacity CAP InNode NO in N)` for the case in which there is no port of the needed capacity `CAP` in one of the nodes traversed by the connection. For simplicity we assume that if there are ports of a given capacity in a node of an upper layer, then there are ports of the same or greater capacity on the nodes of the layers below.

The protocol followed by the modification process is sketched in Figure 3. First, messages are sent to the connection, links and nodes involved in the requested change, to verify whether the modification is possible. If the returned messages indicate that the modification can indeed be done, then the modification process starts changing the service demand on the connection, the bandwidth required on the links that support the connection, and the number of ports of the traversed nodes; otherwise, the corresponding error message is sent to the external object `O` through the network object.

The network object `N` forwards the query to its set of connections, using the message `ComMod(O,N,NO1,NO2,CS,<<S;D>>)` by means of the rule.

```
rl ChDemand(O,N,NO1,NO2,<<S;D>>)
   < N : C-Network | NodeSet: NO1 NO2 NS, ConnectionSet: CS >
=> < N : C-Network > ComMod(O,N,NO1,NO2,CS,<<S;D>>) .
```

where the variables are:

```
var O : OId .                    var D : Nat .
var N : NetworkOId .             var NS : NodeOIdSet .
vars NO1 NO2 : NodeOId .         var CS : ConnectionOIdSet .
var S : ServiceOId .
```

The set of connections is explored one connection at a time. If a connection between the required nodes NO1, NO2 is found, then a new message is sent to the list of links that form the connection to verify whether the modification is possible; otherwise, an error message is sent to the network object N, which forwards it to the external object O. A different and more complex approach could be to create the connection when it does not exist. The corresponding rules are

```
rl ComMod(O,N,NO1,NO2,C CS,<<S;D>>)
   < C : C-Connection | Nodes: <<M1;M2>>, LinkList: LL >
   < S : C-Service | Capacity: CP >
=> < C : C-Connection > < S : C-Service >
   LinkListLoadReq(O,N,C,<<S;D>>,LL,D*CP)
   if ((M1 == NO1) and (M2 == NO2)) or
      ((M2 == NO1) and (M1 == NO2)) .

rl ComMod(O,N,NO1,NO2,C CS,<<S;D>>)
   < C : C-Connection | Nodes: <<M1;M2>> >
=> < C : C-Connection > ComMod(O,N,NO1,NO2,CS,<<S;D>>)
   if ((M1 =/= NO1) or (M2 =/= NO2)) and
      ((M2 =/= NO1) or (M1 =/= NO2)) .

rl ComMod(O,N,NO1,NO2,null,<<S;D>>)
=> (To N and O NoConnectionBetweenNodes NO1 to NO2) .

rl (To N and O NoConnectionBetweenNodes NO1 to NO2)
   < N : C-Network >
=> < N : C-Network >
   (To O NoConnectionBetweenNodes NO1 to NO2 in N) .
```

where the variables not declared previously are:

```
vars M1 M2 : NodeOId .
var LL : LinkOIdList .
var CP : Nat .
```

If a connection between the required nodes is found in the network, the verification process continues with the `LinkListLoadReq(O,N,C,<<S;D>>,LL,R)` message, which goes over the list of links that form the connection, sending messages to its ending nodes to verify their port capacity. The rules are

```
vars L L1 : LinkOId .
var R : Nat .

rl LinkListLoadReq(O,N,C,<<S;D>>,L LL,R)
   < L : C-Link | Nodes: <<N1;N2>> >
=> < L : C-Link > PortNodeReq(O,N,C,L,<<S;D>>,N1,R)
   PortNodeReq(O,N,C,L,<<S;D>>,N2,R)
   LinkListLoadReq2(O,N,C,<<S;D>>,LL,L,N1,N2,R) .

rl LinkListLoadReq2(O,N,C,<<S;D>>,L LL,L1,N1,N2,R)
   (To L1 and O and N and C and <<S;D>> PortInNode N1)
   (To L1 and O and N and C and <<S;D>> PortInNode N2)
   < L : C-Link | Nodes: <<M1;M2>> >
=> < L : C-Link > PortNodeReq(O,N,C,L,<<S;D>>,M1,R)
   PortNodeReq(O,N,C,L,<<S;D>>,M2,R)
   LinkListLoadReq2(O,N,C,<<S;D>>,LL,L,M1,M2,R) .

rl LinkListLoadReq2(O,N,C,<<S;D>>,nil,L,N1,N2,R)
   (To L and O and N and C and <<S;D>> PortInNode N1)
   (To L and O and N and C and <<S;D>> PortInNode N2)
=> (To C and O and N and <<S;D>> LinksVerified) .
```

A new message `LinkListLoadReq2` is used to go over the list and summarize the acknowledgement message of the ending nodes. This auxiliary message is used only for the implementation of this process and it will not be invoked by any other object. In this case, an encapsulation mechanism would be very useful, as it would avoid the possible misuse of this message by any other object.

Finally, the message `PortNodeReq(O,N,C,L,<<S;D>>,NO,R)` is sent to the node `NO` to verify whether it has ports to carry at least a bandwidth `R`, thus finishing the verification process. The implementation of this rule is different in the physical layer from the other two layers, because of the different equipment defined in each kind of node. For this reason the rules are implemented in the specialized classes of each layer and not in the generic classes like previous rules. The rules for physical nodes are:

```
var NO : NodeOId .
var ET : EquipmentOId .
vars U CAP : Nat .

rl PortNodeReq(O,N,C,L,<<S;D>>,NO,R)
   < NO : C-Physical-Node |
              EqTrans: <ET : C-Eq-Trans | Capacity: CAP >,
              UsedPorts: U >
   < S : C-Service | Capacity: CP >
=> < NO : C-Physical-Node > < S : C-Service >
   (To L and O and N and C and <<S;D>> PortInNode NO)
   if (CP <= CAP) and (U + R DIV CAP >= 0) .
```

```
rl PortNodeReq(O,N,C,L,<<S;D>>,NO,R)
   < NO : C-Physical-Node |
                EqTrans: < ET : C-Eq-Trans | Capacity: CAP >
                UsedPorts: U >
   < S : C-Service | Capacity: CP >
=> < NO : C-Physical-Node > < S : C-Service >
   (To N and O NoPortCapacity R Node NO)
   if (CP > CAP) or (U + R DIV CAP < 0) .
```

The rules check that the port capacity CAP is enough to carry one communication and that the number of ports in the node will not be less than zero after the modification. The last condition is necessary because the modification can decrease the number of communications on the connection. Notice that the value of the EqTrans attribute is a configuration consisting of a transmission equipment object instead of an object identifier. The reason is that a containment relationship has been declared between the nodes and the associated equipment, and this relationship is implemented using configurations as the value of the corresponding attributes as it is explained in Section 5.2.

Once it is verified that all the links and nodes in the connection can support the change of demand, the connection object starts the modification process by means of the message ComMod2(O,N,C,<<S;D>>,DL)

```
rl (To C and O and N and <<S;D>> LinksVerified)
   < C : C-Connection | DemandList: DL >
=> < C : C-Connection > ComMod2(C,<<S;D>>,DL)
   (To N and O ConnectionModified C) .
```

where DL denotes a DemandList, that is, a list of tuples, each one consisting of a service identifier and the number of communications of this service carried by the connection.

The ComMod2 message goes over the demand list looking for a tuple whose service identifier is the one used in the message. If it exists, the demand is changed; otherwise, the service is added to the demand list. In both cases a message to change the links is sent.

```
vars S S1 S2 : ServiceOId .          vars D1 D2 : Nat .
vars DL1 DL2 : DemandList .

rl ComMod2(C,<<S;D1>>,<<S;D2>> DL )
   < C : C-Connection |
         DemandList: DL1 <<S;D2>> DL2, LinkList: LL >
   < S : C-Service | Capacity: CP >
=> < C : C-Connection | DemandList: DL1 <<S;D1+D2>> DL2 >
   < S : C-Service > ChLinkListLoad(LL,D1*CP) .

rl ComMod2(C,<<S1;D1>>,<<S2;D2>> DL) < C : C-Connection >
=> < C : C-Connection > ComMod2(C,<<S1;D1>>,DL)
   if (S1 =/= S2) .
```

```
rl ComMod2(,C,<<S;D1>>,nil)
   < C : C-Connection | DemandList: DL, LinkList: LL >
   < S : C-Service | Capacity: CP >
=> < C : C-Connection | DemandList: DL <<S;D1>> >
   < S : C-Service > ChLinkListLoad(LL,D1*CP) .
```

The message `ChLinkListLoad(LL,R)` changes the load of the links in the list `LL` adding the bandwidth `R`. Like in the `PortNodeReq` message, the rules that implement this message in the physical layer are different from the rules in the other two layers. In the former they are

```
var LD : Nat .

rl ChLinkListLoad(L LL,R)
   < L : C-Physical-Link | Nodes: <<N1;N2>>, Load: LD >
=> < L : C-Physical-Link | Load: LD + R >
   ChPortNodes(N1,R) ChPortNodes(N2,R)
   ChLinkListLoad(LL,R) .

rl CHLinkListLoad(nil,R) => nil .
```

Finally, the message `ChPortNodes(NO,R)` changes the number of ports used in node `NO` in accordance with the new bandwidth `R`. Once more, the rule for the physical layer is different from the rule for the other two layers.

```
rl ChPortNodes(NO,R)
   < NO : C-Physical-Node |
                EqTrans: < ET : C-Eq-Trans | Capacity: CAP >,
                Used: U >
=> < NO : C-Physical-Node | Used: U + R DIV CAP > .
```

5 Implementation of the object relationships

Three relationships are considered in the database model: the inheritance relationship, the containment relationship, and explicit group relationships. The inheritance relationship is directly supported by Maude, which, as we have mentioned in Section 2, distinguishes two kinds of inheritance: class inheritance and module inheritance. On the contrary, containment and explicit group relationships are not directly supported by the language, although they can be specified in a natural way within it.

5.1 Inheritance relationship

Class inheritance is defined between generic classes `C-Network`, `C-Node`, `C-Link` and `C-Connection`, and the classes specialized for each layer.

In the case of nodes, a new class `C-ATM-Node` is defined to capture the common behavior of the two classes `C-VP-Node` and `C-VC-Node`. Basically, it implements the cost messages which are computed in a different way for the

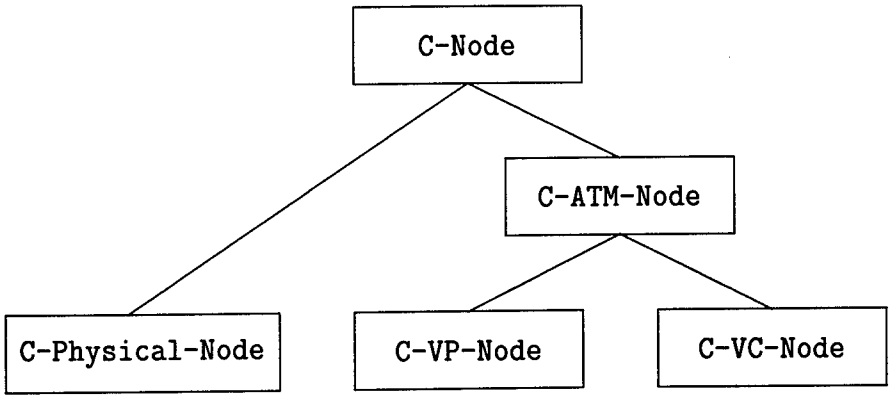


Fig. 4. Complete node inheritance relationship

physical layer and for the other two layers because the equipment is different in each case. The class inheritance relationship is then as represented in Figure 4.

As the node cost messages must be implemented also in the module for the class `C-Physical-Node`, we could use module inheritance to redefine them from the module defining the class `C-ATM-Node`. However, we have chosen not to do so because almost all of the original messages need to be redefined, thus defying the purpose of code reuse. Instead, we simply provide the new rules for the corresponding messages in the module defining the class `C-Physical-Node`.

5.2 The containment relationship

We compare two approaches to the implementation of the containment relationship. The first one considers the OIds of the contained objects as attributes of the owner object. The second one defines an attribute of the owner object as a subconfiguration consisting of the contained objects [8,2]. In both cases, the containment relationship impacts on the object creation and deletion rules of Maude. Object creation should now not only ensure the uniqueness of object identity, but also the two properties of the containment relationship:

- The contained object must have a unique owner object.
- The contained object cannot exist if its owner object does not exist.

Object creation was defined in [8] by means of `new(C | ATTS)` messages where `C` is the name of the class and `ATTS` are the attributes. In order to impose uniqueness of object identity, it is assumed that in the initial configuration we have a collection of different objects of the class `ProtoObject` which has an attribute `counter`, whose value is a natural number used as part of the new object name. The rules for creating objects have the form

```
rl new(C | ATTS)  < O : ProtoObject | counter: N >
=> < O : ProtoObject | counter: N+1 > < O.N : C | ATTS > .
```

This scheme has been slightly modified in [9] to force some attributes of the object to always have some fixed initial values. This is done by declaring

an `initially` clause in the class definition which states the attributes that have an initial value as well as that value. For example, we can have the following class declaration:

```
class X | a1: t1, a2: t2, a3: t3, a4: t4 .
initially a1: v1, a2: v2 .
```

where `ai` are the attribute identifiers, `ti` are the attribute types, and `vi` are the attribute values. The rules for creating objects have now the form

```
rl new(C | a3: v3, a4: v4)  < O : ProtoObject | counter: N >
=> < O : ProtoObject | counter: N+1 >
   < O.N : C | a1: v1, a2: v2, a3: v3, a4: v4 > .
```

The first approach to the implementation of the containment relationship declares the identifiers of contained objects as attributes of the owner object, and following the previous scheme we can force them to have an initial value in the creation process. For example, the creation rules for a class `X` with contained classes `Y1` and `Y2` could be defined as follows:

```
class Y1 | b1: t1, b2: t2 .
initially b1: v1, b2: v2 .

class Y2 | c1: t1', c2: t2' .
initially c1: w1, c2: w2 .

class X | a1: OId, a2: OId, a3: t3, a4: t4 .
initially a1: Y1, a2: Y2 .

rl new(X | a3: v3, a4: v4) < O : ProtoObject | counter: N >
=> < O : ProtoObject | counter: N+1 >
   < O.N : X | a1: O.N.1, a2: O.N.2, a3: v3, a4: v4 >
   < O.N.1 : Y1 | b1: v1, b2: v2 >
   < O.N.2 : Y2 | c1: w1, c2: w2 > .
```

The result of applying this rule is that the `new` message dissapears, the attribute `counter` of the `ProtoObject` object is incremented, and three new objects are created in the system: an object of the class declared on the `new` message, whose identifier is unique in the system because it is formed automatically with the object identifier of the `ProtoObject` object and the value of the `counter` attribute, and with two attributes that are object identifiers. These two object identifiers are also unique in the system as they are formed with the owner object identifier. The other two objects correspond to the contained objects. This kind of rule guarantees that the contained objects are created with the owner object, and the naming process is the one used in telecommunication networks.

The main differences introduced in this creation process with respect to the previous one are:

- The `initially` clause defines the class of the contained objects instead of the initial value of the attribute. The object identifier is given automatically

by the creation rule. It consists of the combination of the owner object identifier and an identifier given to each contained object by the owner object (in the example above, a number associated to the attribute name).

- The rule for the `new` message creates the new object as well as the contained objects. Creating the contained objects directly instead of sending more `new` messages allows us to name the contained objects using the name of the owner object as it is usual in telecommunication networks. By sending `new` messages, the object would instead be created using the `ProtoObject` class and would therefore have a general identifier.
- No `new` message is provided for the contained objects because they cannot be created outside the creation process of their owner objects.

The approach presented in [8] to object deletion uses a message `delete(A)`, with a rule

```
rl delete(A) < A : X | ATTS > => null .
```

To maintain the containment relationship, contained objects should be deleted at the time the owner object is deleted. This is simply achieved by sending deletion messages to all contained objects in the owner object deletion rule.

The second approach, which treats subobjects as part of the owner's object state, is simpler. The previous creation rule is defined in this case by:

```
class Y1 | b1: t1, b2: t2 .
initially b1: v1, b2: v2 .

class Y2 | c1: t1', c2: t2' .
initially c1: w1, c2: w2 .

class X | conf: Subconfiguration of Y1 Y2, a3: t3, a4: t4 .

rl new(X | a3: v3, a4: v4)  < O : ProtoObject | counter: N >
=> < O : ProtoObject | counter: N+1 >
   < O.N : X | conf: < O.N.1 : Y1 | b1: v1, b2: v2 >
                     < O.N.2 : Y2 | c1: w1, c2: w2 >,
               a3: v3, a4: v4 > .
```

In our proposed model, we have selected the second approach, as it is directly supported by Maude without any other consideration. As previously mentioned, the class `C-ATM-Node` defines a containment relationship between the node and the equipment:

```
class C-ATM-Node | EqConm: Subconfiguration of C-Eq-Conm,
                   UtilL: EQCUtilList .
```

Nevertheless, the first approach could be more elegant when more containment relationships are considered between the network objects.

5.3 *Explicit group relationships*

Explicit group relationships are modeled directly by defining a set of object identifiers as the value of an attribute. The only restriction imposed on the object creation process is that the objects related to the attribute value should also exist in the database. Following this idea, a network object, related by a "member-of" relationship to the set of nodes, links and connections that form it, is initially created with empty sets of elements. Then the nodes, links and connections are added to the network using messages `AddNode`, `AddLink` and `AddConnection`. The rules implementing these messages create the objects at the same time that they are introduced as elements of the network.

```
rl (AddNodeTo N)
=> (new C-Node ack N req X) AddNodeToNetwork(N,X) .

rl (to N:X is NO) AddNodeToNetwork(N,X)
   < N : C-Network | NodeSet: NS >
=> < N : C-Network | NodeSet: NO NS > .
```

The node object is created using the `new` message because the node identifier is given in the system name space controlled by the object of class `ProtoObject`, as the explicit group relationships do not create a naming space like the containment relationship. Once the node object has been created in the system, the second rule is used to introduce it in the corresponding network.

In the case of links and connections, which have additional group relationships declared with other objects, the rules also check that all necessary requirements are satisfied. For example, the rules for the message `(AddLinkTo N Between NO1 and NO2)` that adds a link to a physical network must check that the two associated nodes have previously been added to the network.

```
rl (AddLinkTo N Between NO1 and NO2)
   < N : C-Physical-Network | NodeSet: NO1 NO2 NS >
=> < N : C-Physical-Network >
   (new C-Physical-Link ack N req X)
   AddLinkToNetwork(N,X,NO1,NO2) .

rl (to N:X is L) AddLinkToNetwork(N,X,NO1,NO2)
   < N : C-Physical-Network | LinkSet: LS >
   < L : C-Physical-Link >
=> < N : C-Physical-Network | LinkSet: L LS >
   < L : C-Physical-Link | Nodes: <<NO1;NO2>> > .
```

Using this creation process it is guaranteed that the resulting network topology is consistent and there will not be attribute values related to objects that do not exist. Consistency should be maintained in the deletion process by imposing that objects related to other objects cannot be eliminated without deleting previously the value of the attribute on the related object. Notice that in explicit group relationships it is enough to delete the value of an attribute

and it is not necessary to delete the complete object like in the containment relationship.

6 Concluding remarks

The Maude language and rewriting logic are very appropriate for the specification of object-oriented systems as they integrate the algebraic data types with the object classes, and allow the definition of static and dynamic aspects of the database within one language.

An implementation of the telecommunication model was also developed using the C++ language. In general, the main advantages of using Maude over the use of object-oriented languages like C++, are the logically based declarative definition of the language semantics, its integrated support of algebraic data types, and the declarative and implicit use of concurrency. In particular the main problem that arises when using C++ is the representation of the explicit group relationship, because the references to the related objects are implemented using pointers to objects, and this introduces all the problems related to the use of pointers in C. The representation of the containment relationship is done by declaring the contained object as the value of an attribute in the owner object and overloading the creation and deletion methods of the class to create/delete at the same time the contained objects, as it is done in the rules described in Section 5.2.

With respect to the design of the Maude language, in our opinion there are some aspects that need more attention. For example, as we have already remarked, sometimes it is necessary to define internal messages in a class that should not be seen from outside the class. This is not possible in the current design of the language, since all messages receive the same treatment. More generally, it is necessary a more detailed study of the subject of *encapsulation* in the context of Maude.

From a similar point of view, the distinction in Maude between class inheritance and module inheritance is not completely satisfactory, due to the impossibility of redefining attributes or messages in subclasses. On the one hand, this forces the creation of additional subclasses, like for example the class `C-ATM-Node` in our application, and on the other hand creates inheritance relationships between modules that in principle have no reason to exist. Moreover, from a modeling standpoint, it does not allow the existence in a subclass of method specializations that are consistent with real life inheritance classifications.

Acknowledgement

We are very grateful to José Meseguer for his detailed comments on previous versions of this paper.

References

[1] ISO/IEC DIS 10165-1/ITU-TS X.720, *Management Information Model* (1990).

[2] Ulrike Lechner, Christian Lengauer, Friederike Nickl, and Martin Wirsing, (Objects + Concurrency) & Reusability - A proposal to circumvent the inheritance anomaly, in: P. Cointe, ed., *Proc. ECOOP'96, 10th European Conference on Object-Oriented Programming* (LNCS 1098, Springer-Verlag, 1996) 232–247.

[3] Ulrike Lechner, Christian Lengauer, and Martin Wirsing, An object-oriented airport: Specification and refinement in Maude, in: E. Astesiano, G. Reggio, and A. Tarlecki, eds., *Recent Trends in Data Type Specification, 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop, 1994* (LNCS 906, Springer-Verlag, 1995) 351–367.

[4] Luis López Zueros and Isabel Pita, Diseño orientado a objetos aplicado a la gestión integrada de redes, *Proc. Telecom I+D Conference* (Madrid, 1992) 359–368.

[5] Narciso Martí-Oliet and José Meseguer, Rewriting logic as a logical and semantic framework, Technical report SRI-CSL-93-05, Computer Science Laboratory, SRI International (August 1993). To appear in D. M. Gabbay, ed., *Handbook of Philosophical Logic* (Kluwer Academic Publishers).

[6] José Meseguer, Rewriting as a unified model of concurrency, in: J. C. M. Baeten and J. W. Klop, eds., *Proc. CONCUR'90* (LNCS 458, Springer-Verlag, 1990) 384–400.

[7] José Meseguer, Conditional rewriting logic as a unified model of concurrency, *Theoretical Computer Science* **96** (1992) 73–155.

[8] José Meseguer, A logical theory of concurrent objects and its realization in the Maude language, in: G. Agha, P. Wegner, and A. Yonezawa, eds., *Research Directions in Concurrent Object-Oriented Programming* (The MIT Press, 1993) 314–390.

[9] José Meseguer, Solving the inheritance anomaly in concurrent object-oriented programming, in: O. M. Nierstrasz, ed., *Proc. ECOOP'93* (LNCS 707, Springer-Verlag, 1993) 220–246.

[10] José Meseguer, Rewriting logic as a semantic framework for concurrency: A progress report, in: *Proc. CONCUR'96* (LNCS, Springer-Verlag, 1996).

[11] José Meseguer and Xiaolei Qian, A logical semantics for object-oriented databases, in: P. Buneman and S. Jajodia, eds., *Proc. SIGMOD'93* (ACM Press, 1993) 89–98.

[12] José Meseguer and Timothy Winkler, Parallel programming in Maude, in: J.-P. Banâtre and D. Le Mètayer, eds., *Researchs Directions in High-Level Parallel Programming Languages* (LNCS 574, Springer-Verlag, 1992) 253–293.

[13] Barbara Salmansberger, Objektorientierte Spezifikation von verteilten Systemen in Maude am Beispiel eines Flughafens, Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau (December 1993).

List of Authors

Peter Borovanský, INRIA Lorraine and CRIN-CNRS, France
Carlos Castro, INRIA Lorraine and CRIN-CNRS, France
Anna Ciampolini, Universitá di Bologna, Italy
Manuel Clavel, SRI International, USA
Răzvan Diaconescu, JAIST, Japan
Steven Eker, SRI International, USA
Fabio Gadducci, Universitá di Pisa, Italy
Claude Kirchner, INRIA Lorraine and CRIN-CNRS, France
Hélène Kirchner, INRIA Lorraine and CRIN-CNRS, France
Alexander Knapp, Ludwig-Maximilians-Universität München, Germany
Evelina Lamma, Universitá di Bologna, Italy
Christopher Landauer, The Aerospace Corporation, USA
Ulrike Lechner, Universität Passau, Germany
Patrick Lincoln, SRI International, USA
Narciso Martí-Oliet, Universidad Complutense de Madrid, Spain
Paola Mello, Universitá di Bologna, Italy
José Meseguer, SRI International, USA
Hiroyuki Miyoshi, Shukutoku University, Japan
Ugo Montanari, Universitá di Pisa, Italy
Pierre-Etienne Moreau, INRIA Lorraine and CRIN-CNRS, France
Peter C. Ölveczky, SRI International, USA and Univ. of Bergen, Norway
Isabel Pita, Universidad Complutense de Madrid, Spain
W. Marco Schorlemmer, IIIA-CSIC, Campus UAB, Spain
Cesare Stefanelli, Universitá di Bologna, Italy
Carolyn Talcott, Stanford University, USA
Patrick Viry, Universitá di Pisa, Italy
Marian Vittek, INRIA Lorraine and CRIN-CNRS, France
Martin Wirsing, Ludwig-Maximilians-Universität München, Germany

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searc gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate o collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations a Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washi

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 9/6/96	3. REPORT TYPE AND DATES COVERED Proceedings Sept. 3 through Sept. 6, 1007

4. TITLE AND SUBTITLE
Rewriting LOgic and its Applications, First International Workshop, Proceedings

5. FUNDING NUMBERS
ONR Grant N00014-96-1-0824

6. AUTHOR(S)
J. Meseguer, Editor

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

8. PERFORMING ORGANIZATION REPORT NUMBER
ECU 7242

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Program Officer Ralph F. Wachter, Code 311
800 North Quincy St., Arlington, VA 22217-5660

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Published by Elsevier Science B.V. in Electronic Notes in Theoretical Computer Science, Vol. 4 Copyright Elsevier B.V.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This volume constitutes the proceedings of the First International Workshop on Rewriting Logic, held at the Asilomar Conference Center, Pacific Grove, California, Sept. 2-6, 1996. The three invited papers by Narciso Marti-Oliet, Ugo Montanari, and Martin Wirsing, and seventeen contributed papers selected by the Program Committee give a rich view of the latest developments and research directions in the field of rewriting logic and its various applications to computing. Besides work on models and on concurrency aspects, there are several papers describing the different rewriting logic languages developed so far in Europe and the U.S., as wellas a paper on semantic foundations for the Cafe language in Japan. There are also several papers on logical and metalogical specification; on reflection and strategies; on applications to object-oriented design, specification and programming; and on applications to contraint solving, to real-time systems, and to discrete event simulation. In addition to the papers, the workshop placed strong emphasis on facilitating in-depth discussion of the topics by allowing ample time for the discussion of each paper, and by including five panel discussions and a tutorial.

14. SUBJECT TERMS
Rewriting Logic

15. NUMBER OF PAGES 426

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited