Best
Available
Copy

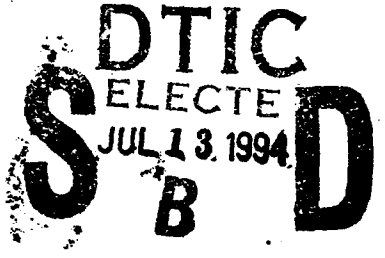
AD-A281 497

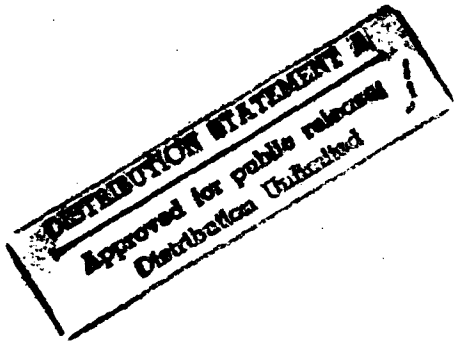


(0)

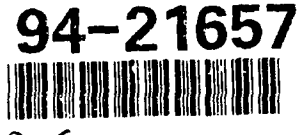
Position Papers for the First Workshop on Principles and Practice of Constraint Programming

April 28–30, 1993
Newport, Rhode Island





94-21657



324pp

Sponsored in part by the Office of Naval Research

Organized by Brown University

Reference materials for workshop participants only—Do not distribute

94 7 12 436

DTIC QUALITY INSPECTED 1

Position Papers for the First Workshop on Principles and Practice of Constraint Programming

April 28–30, 1993
Newport, Rhode Island

Sponsored in part by the Office of Naval Research

Organized by Brown University

Reference materials for workshop participants only—Do not distribute

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By per form 50
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Contents

Hassan Aït-Kaci and Andreas Podelski (DEC Paris).
Entailment and Disentailment of Order-Sorted Feature Constraints 1
Alexander Brodsky and Catherine Lassez (IBM Watson).
Separability of Polyhedra and a New Approach to Spatial Storage 6
Allen L. Brown Jr., Surya Mantha, and Toshiro Wakayama (Xerox SRL).
Constraint Optimization using Preference Logics: A New Role for Modal Logic 14
Isabel F. Cruz (Brown University).
Using a Visual Constraint Language for Data Display Specification 24
Stéphane Donikian (IRISA) and Gérard Hégron (Ecole des Mines de Nantes).
Constraint Management in a Declarative Design Method for 3D Scene Sketch Modeling 36
Thomas Dubé and Chee-Keng Yap (New York University).
The Geometry in Constraint Logic Programs 46
François Fages (LIENS and LCR Thomson).
On the Semantics of Optimization Predicates in CLP Languages 53
Tim Fernando (CWI Amsterdam).
A Higher-Order Extension of Contraint Programming in Discourse Analysis 62
Eugene C. Freuder and Paul D. Hubbe (University of New Hampshire).
A Disjunctive Decomposition Control Schema for Constraint Satisfaction 72
Thom Frühwirth (ECRC München) and Philipp Hanschke (DKFI Kaiserslautern).
Terminological Reasoning with Constraint Handling Rules 82
Hong Gao and David S. Warren (SUNY at Stony Brook).
A Powerful Evaluation Strategy for CLP Programs 92
Michael Gleicher (Carnegie Mellon University).
Practical Issues in Programming Constraints 100
Seif Haridi, Sverker Janson, Johan Montelius, Torkel Franzén, Per Brand, Kent Boortz, Björn Danielsson, Björn Carlson, Torbjörn Keisu, Dan Sahlin and Thomas Sjöland (SICS).
Concurrent Constraint Programming at SICS with the Andorra Kernel Language 109
Jean-Louis Imbert (G.I.A. Parc Scientifique et Technologique de Luminy).
Fourier's Elimination: Which to Choose? 119
Philippe Jégou (Université de Provence).
Domains Decomposition in Finite Constraint-Satisfaction Problems 132
Mark Johnson (Brown University).
Memoization in Constraint Logic Programming 140
Simon Kasif (John Hopkins University) and Arthur L. Delcher (Loyola College).
Local Consistency in Parallel Constraint-Satisfaction Networks 149
Walid T. Keirouz, Glenn A. Kramer, and Jahir Pabon (Schlumberger Laboratories).
Exploiting Constraint Dependency Information For Debugging and Explanation 156

Claude Kirchner, Hélène Kirchner, and Marian Vittek (INRIA Lorraine and CRIN).
Implementing Computational Systems with Constraints 166
Gabriel M. Kuper (ECRC München).
Aggregation in Constraint Databases 176
François Major (National Institutes of Health), Marcel Turcotte, and Guy Lapalme (Université de Montréal).
Constraint Satisfaction in Functional Programming 184
Ken McAloon and Carol Tretkoff (CUNY and Brooklyn College).
2lp: Linear Programming and Logic Programming 189
Francisco Menezes, Pedro Barahona (Universidade Nova de Lisboa), and Philippe Codognet (INRIA Rocquencourt).
An Incremental Hierarchical Constraint Solver 201
Scott Meyers, Carolyn K. Duby, and Steven P. Reiss (Brown University).
Constraining the Structure and Style of Object-Oriented Programs 211
Spiro Michaylov (Ohio State University) and Frank Pfenning (Carnegie Mellon University).
Higher-Order Logic Programming as Constraint Logic Programming 221
Ugo Montanari and Francesca Rossi (Università di Pisa).
Constraint Satisfaction, Constraint Programming, and Concurrency 230
William J. Older (Bell Northern Research) and Frédéric Benhamou (Faculté des Sciences de Luminy).
Programming in CLP(BNR) 239
Dinesh K. Pai (University of British Columbia).
Robot Programming and Constraints 250
William C. Rounds and Guo-Qiang Zhang (University of Michigan, Ann Arbor).
Constraints In Nonmonotonic Reasoning 258
Michael Sannella (University of Washington, Seattle).
The SkyBlue Constraint Solver and Its Applications 268
Tony Savor and Paul Dasiewicz (University of Waterloo).
A Real Time Extension to Logic Programming Based on the Concurrent Constraint Logic Programming Paradigm 279
Douglas R. Smith (Kestrel Institute).
Synthesis of Constraint Algorithms 288
Terence R. Smith and Keith Park (University of California, Santa Barbara).
Constraint-Based Languages for Scientific Database and Modeling Systems 294
Allen C. Ward (University of Michigan, Ann Arbor).
Set-based Concurrent Engineering 299
Ying Zhang and Alan K. Mackworth (University of British Columbia).
Constraint Programming in Constraint Nets 303
Richard Zippel (Cornell University).
A Constraint Based Scientific Programming Language 313

Author Index **319**

Entailment and Disentailment of Order-Sorted Feature Constraints

(Summary)*

Hassan Aït-Kaci Andreas Podelski
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92500 Rueil-Malmaison, France
{hak,podelski}@prl.dec.com

Abstract

LIFE uses matching on order-sorted feature structures for passing arguments to functions. As opposed to unification which amounts to normalizing a conjunction of constraints, solving a matching problem consists of deciding whether a constraint (guard) or its negation are entailed by the context. We give a complete and consistent set of rules for entailment and disentailment of order-sorted feature constraints. These rules are directly usable for relative simplification, a general proof-theoretic method for proving guards in concurrent constraint logic languages using guarded rules.

1 Introduction

LIFE [5, 4] extends the computational paradigm of Logic Programming in two essential ways:

- using a data structure richer than that provided by first-order constructor terms; and,
- allowing interpretable functional expressions as *bona fide* terms.

The first extension is based on ψ-terms which are attributed partially-ordered sorts denoting sets of objects [1, 2]. In particular, ψ-terms generalize first-order constructor terms in their rôle as data structures in that they are endowed with a unification operation denoting type intersection.

The second extension deals with building into the unification operation a means to reduce functional expressions using definitions of interpretable symbols over data patterns. The basic insight is that unification is no longer seen as an atomic operation by the resolution rule. Indeed, since unification amounts to normalizing a conjunction of equations, and since this normalization process commutes with resolution, these equations may be left in a normal form that is not a fully solved form. In particular, if an equation involves a functional expression whose arguments are not sufficiently instantiated to match a *definiens* of the function in question, it is simply left untouched. Resolution may proceed until the arguments are *proven* to match a definition from the accumulated constraints in the context [3]. This simple idea turns out invaluable in practice.

This technique—delaying reduction and enforcing determinism by allowing only equivalence reductions—is called *residuation* [3]. It does not have to be limited to functions. Therefore, we explain it for the general case of relations. Intuitively, the arguments of a relation which are constrained by the guard are its input parameters and correspond to the arguments of a function. This has been used as an implicit control mechanism in general concurrent constraint logic programming schemes; *e.g.*, the logic of guarded Horn-clauses studied by Maher [9], Concurrent Constraint Programming (CCP) [10], and Kernel Andorra Prolog (KAP) [8]. These schemes are parameterized with respect to an abstract class of constraint systems. An incremental test for entailment and disentailment between constraints is needed for advanced control mechanisms such as delaying, coroutining, synchronization, committed choice, and deep constraint propagation. LIFE is formally an instance of this scheme, namely a CLP language

*Full version to appear in [6].

using a constraint system based on order-sorted feature (OSF) structures [5]. It employs a related, but limited, suspension strategy to enforce deterministic functional application. Roughly, these systems are concurrent thanks to the new effective discipline for procedure parameter-passing that can be described as "call-by-constraint-entailment" (as opposed to Prolog's call-by-unification).

The most direct way to explain the issue is with an example. In LIFE, one can define functions as usual; say:

$$\begin{aligned} fact(0) &\rightarrow 1. \\ fact(N : int) &\rightarrow N * fact(N - 1). \end{aligned}$$

More interesting is the possibility to compute with partial information. For example:

$$\begin{aligned} minus(negint) &\rightarrow posint. \\ minus(posint) &\rightarrow negint. \\ minus(zero) &\rightarrow zero. \end{aligned}$$

Let us assume that the symbols *int*, *posint*, *negint*, and *zero* have been defined as sorts with the approximation ordering such that *posint*, *zero*, *negint* are pairwise incompatible subsorts of the sort *int* (*i.e.*, $posint \wedge zero = \bot$, $negint \wedge zero = \bot$, $posint \wedge negint = \bot$). This is declared in LIFE as $int := \{posint; zero; negint\}$. Furthermore, we assume the sort definition $posint := \{posodd; poseven\}$; *i.e.*, *posodd* and *poseven* are subsorts of *posint* and mutually incompatible.

The LIFE query $Y = minus(X : poseven)$? will return $Y = negint$. The sort *poseven* of the actual parameter is incompatible with the sort *negint* of the formal parameter of the first rule defining the function *minus*. Therefore, that rule is skipped. The sort *poseven* is more specific than the sort *posint* of the formal parameter of the second rule. Hence, that rule is applicable and yields the result $Y = negint$.

The LIFE query $Y = minus(X : string)$ will fail. Indeed, the sort *string* is incompatible with the sort of the formal parameter of every rule defining *minus*.

Thus, in order to determine which of the rules, if any, defining the function in a given functional expression will be applied, two tests are necessary:

- verify whether the actual parameter is more specific than or equal to the formal parameter;
- verify whether the actual parameter is at all compatible with the formal parameter.

What happens if both of these tests fail? For example, consider the query consisting of the conjunction:

$$Y = minus(X : int), X = minus(zero)?$$

Like Prolog, LIFE follows a left-to-right resolution strategy and examines the equation $Y = minus(X : int)$ first. However, both foregoing tests fail and deciding which rule to use among those defining *minus* is inconclusive. Indeed, the sort *int* of the actual parameter in that call is neither more specific than, nor incompatible with, the sort *negint* of the first rule's formal parameter. Therefore, the function call will *residuate* on the variable X. This means that the functional evaluation is suspended pending more information on X. The second goal in the query is treated next. There, it is found that the actual parameter is incompatible with the first two rules and is the same as the last rule's. This allows reduction and binds X to *zero*. At this point, X has been instantiated and therefore the residual equation pending on X can be reexamined. Again, as before, a redex is found for the last rule and yields $Y = zero$.

The two tests above can in fact be worded in a more general setting. Viewing data structures as constraints, "more specific" is simply a particular case of constraint entailment. We will say that a constraint *disentails* another whenever their conjunction is unsatisfiable; or, equivalently, whenever it entails its negation. In particular, first-order matching is deciding entailment between constraints consisting of equations over first-order terms. Similarly, deciding unifiability of first-order terms amounts to deciding "compatibility" in the sense used informally above.

The suspension/resumption mechanism illustrated in our example is repeated each time a residuated actual parameter becomes more instantiated from the context; *i.e.*, through solving other parts of the query. Therefore, it is most beneficial for a practical algorithm testing entailment and disentailment to be incremental. This means that, upon resumption, the test for the instantiated actual parameter builds upon partial results obtained by the previous test. One outcome of the results presented in this paper is that it is possible to build such a test; namely, an algorithm deciding simultaneously two problems in an incremental manner—entailment and disentailment. The technique that we have devised to do that is called *relative simplification* of constraints [4, 7].

Besides incrementality, the relative-simplification technique has the advantage of yielding, in case of entailment, the instantiation of the formal parameter by the actual parameter, as we will explain next.

Every guarded language produces a new environment, namely the conjunction of the old environment, which is the constraint part of the resolvent (the context), and the guard. This conjunction affects the variables in the body (*viz.*, in LIFE, the right-hand side expression of a function definition) after successfully executing the corresponding guard; *i.e.*, it "constrains" them in a semantical sense.

For example, if (in the Herbrand constraint system) $Y = f(a)$ is the context and $Y = f(X)$ is the guard and $Z = X$ is the body, then X is constrained to be equal to a. Practically, the matching proof is done by unification which yields the *instantiation* of the body variable X, $X = a$. In order to compute the new environment, this unification is, of course, not repeated.

The example above can be extended to OSF constraint systems. Thanks to our method, the proof of entailment has as a consequence (somewhat like a side-effect) that the conjunction of the context and the guard is in solved form, as if normalized by the OSF constraint solver. Now, in this solved form, the formal variables are bound to the global ones. This is what we mean by the instantiation of the formal parameter by the actual parameter.

2 OSF Formalism

The syntax and semantics of the formulas that we use as constraints is fixed by an *order-sorted feature signature* (or simply OSF signature) which is: (1) a set of sorts $\mathcal{S}$, equipped with partial order $\leq$ and meet operation $\wedge$, and (2) a set of features $\mathcal{F}$. A logical structure fitting such a signature (*i.e.*, interpreting sorts as sets, $\leq$ as $\subseteq$, $\wedge$ as $\cap$, and features as unary functions) is called an *OSF algebra*.

An OSF constraint ϕ is a conjunction of formulas of one of the forms: (1) $X : s$, (2) $X \doteq X'$, or (3) $X.\ell \doteq X'$, where X and X' are variables from a given set of variables $\mathcal{V}$, s is a sort in $\mathcal{S}$, and ℓ is a feature in $\mathcal{F}$. The interpretation of ϕ in an OSF algebra $\mathcal{A}$ under a valuation $\alpha : \mathcal{V} \mapsto D^{\mathcal{A}}$, written $\mathcal{A}, \alpha \models \phi$, is as usual.

The set of *OSF terms* is generated with the following context-free rules:

$$t ::= X : s(\ell_1 \Rightarrow t, \ldots, \ell_n \Rightarrow t)$$

where X is a variable from a set $\mathcal{V}$, s is a sort in $\mathcal{S}$, and $\ell_i \in \mathcal{F}$, $n \geq 0$. The variable X is called the term's root variable, the sort s its root sort.

Any OSF term t is equivalently expressible as an OSF clause, denoted $\phi(t)$, called its dissolved form. That is, its meaning $[\![t]\!]^{\mathcal{A}}$ in the OSF algebra $\mathcal{A}$ can be described as the set of all values $\alpha(X)$ for the root X of t such that $\phi(t)$ is satisfied in $\mathcal{A}$ under some valuation α; *i.e.*,

$$[\![t]\!]^{\mathcal{A}} = \{\alpha(X) \mid \alpha : \mathcal{V} \mapsto D^{\mathcal{A}},\ \mathcal{A}, \alpha \models \phi(t)\}.$$

We will often deliberately confuse a ψ-term ψ with its dissolved form $\phi(\psi)$ and refer to $\phi(\psi)$ simply as ψ.

Syntactically consistent OSF term are said to be in normal form, and called ψ-terms. They comprise a set called Ψ. By extension, $\leq$ and $\wedge$ are extended from the sort signature to the set Ψ, realizing matching and unification, respectively.

Unification of OSF terms is done thanks to a normalization procedure. Namely, ψ_1 and ψ_2 are dissolved, and then the OSF constraint $\psi_1 \ \& \ \psi_2 \ \& \ Root(\psi_1) \doteq Root(\psi_2)$ is normalized (into $\bot$ if and only if ψ_1 and ψ_2 are non-unifiable). The rules to normalize OSF terms are not given here.

We obtain one important example of an OSF algebra directly from the syntactic expressions of ψ-terms: the OSF algebra Ψ of ψ-terms. The domain of Ψ is the set of all ψ-terms, up to graph representation. That is, we identify ψ-terms as values of Ψ if they are represented by the same graph. For example, the two ψ-terms $Y : s(\ell_1 \Rightarrow X : s', \ell_2 \Rightarrow X)$ and $Y : s(\ell_1 \Rightarrow X, \ell_2 \Rightarrow X : s')$ correspond to the same object.

A sort $s \in \mathcal{S}$ is interpreted as the set of all ψ-terms whose root sort is a subsort of s. A feature $\ell \in \mathcal{F}$ is interpreted as a function $\ell^{\Psi} : D^{\Psi} \mapsto D^{\Psi}$ which, roughly, maps a ψ-term on its sub-ψ-term accessable by the feature ℓ. For example, taking $\psi = X : \top(\ell_1 \Rightarrow Y : s, \ell_2 \Rightarrow X)$, we have $\ell_1^{\Psi}(\psi) = Y : s$, $\ell_2^{\Psi}(\psi) = \psi$, and $\ell_3^{\Psi}(\psi) = Z_{\ell_3,\psi} : \top$.

According to the triple existence of ψ-terms being set-denoting types, OSF constraints and, as elements of an OSF algebra, concrete data structures, we define three orderings on ψ-terms.

A ψ-term ψ is *subsumed by* a ψ-term ψ' if and only if the denotation of ψ is contained in that of ψ' in all interpretations. Formally,

$$\psi \leq \psi' \ \textit{iff} \ [\![\psi]\!]^{\mathcal{A}} \subseteq [\![\psi']\!]^{\mathcal{A}}$$

for all OSF algebras $\mathcal{A}$.

An approximation preorder $\sqsubseteq$ on ψ-terms is defined such that, ψ_1 *approximates* ψ_2 if and only if ψ_2 is an endomorphic image of ψ_1. Formally, $\psi_1 \sqsubseteq_{\mathcal{A}} \psi_2$ iff $\gamma(\psi_1) = \psi_2$ for some homomorphism $\gamma : \mathcal{A} \mapsto \mathcal{A}$. (A homomorphism between OSF algebras is a mapping between their domains which is compatible with the ordering on sorts and feature application.)

We note that, if we represent ψ-terms as graphs, endomorphisms on Ψ are graph homomorphisms with the additional sort-compatibility property. A node labeled with sort s is always mapped into a node labeled with s or a subsort of s. An edge labeled with a feature is mapped into an edge labeled with the same feature.

Thus, endomorphic approximation captures exactly object-oriented class inheritance. Indeed, if an attribute is present in a class, then it is also present in a subclass with a sort that is the same or refined. Since features are total functions, this also takes care of introducing a new attribute in a subclass: it refines $\top$. Note also, that the restriction of γ to the set of nodes defines a variable binding; it corresponds to the notion of a matching substitution for first-order terms.

A ψ-term ψ *entails* a ψ-term ψ' if and only if, as constraints, ψ implies the conjunction of ψ' and $X \doteq X'$; more precisely,

$$\psi \succeq \psi' \quad \textit{iff} \quad \models \psi \rightarrow \exists \mathcal{U}\, (X \doteq X' \,\&\, \psi')$$

where X, X' are the roots of ψ and ψ' and $\mathcal{U} = Var(\psi')$.

The following proposition states what we call the *semantic transparency of orderings*.

Proposition 1 *The following are equivalent:*

- $\psi \sqsubseteq \psi'$ — *ψ approximates ψ';*
- $\psi' \leq \psi$ — *ψ' is a subtype of ψ;*
- $\psi' \succeq \psi$ — *ψ entails ψ'.*

3 Proving OSF Guards

In the following, we use ϕ as the *context* formula. It is assumed to be satisfiable.

The variables in ϕ are *global*. We shall use $\mathcal{X}$ to designate the set of global variables $Var(\phi)$ and the letters $X, Y, Z, \ldots$, for variables in $\mathcal{X}$. We use ψ, a dissolved ψ-term, as the *guard* formula. The variables in ψ are *local* to ψ; *i.e.*, $Var(\phi) \cap Var(\psi) = \emptyset$. We shall use $\mathcal{U}$ to designate the set of local variables $Var(\psi)$ and the letters $U, V, W, \ldots$, for variables in $\mathcal{U}$. The letter U will always designate the root variable of ψ. We also refer to ϕ as the *actual* parameter, and to ψ as the *formal* parameter.

We investigate a proof system which decides two problems simultaneously:

- the validity of $\forall \mathcal{X}\, \big(\, \phi \rightarrow \exists \mathcal{U}.\, (\psi \,\&\, U \doteq X) \,\big)$;
- the unsatisfiability of $\phi \,\&\, \psi \,\&\, U \doteq X$.

The first test is called a test for *entailment* of the guard by the context, and the second, a test for *disentailment*. This second test is equivalent to testing the validity of the implication $\forall \mathcal{X}\, \big(\, \phi \rightarrow \neg \exists \mathcal{U}.\, (\psi \,\&\, U \doteq X) \,\big)$.

Since both tests amount to deciding whether the context implies the guard or its negation, all local variables are existentially quantified and all global variables are universally quantified.

The *relative-simplification* system for OSF constraints is presented in [4, 6] in form of 11 constraint normalization rules (not given here in this summary).

A set of bindings $U_i \doteq X_i$, $i = 1, \ldots, n$ is a *functional binding* if all the variables U_i are mutually distinct.

The effectuality of the relative-simplification system is summed up in the following statement:

> **Effectuality of Relative-Simplification** *The solved OSF constraint ϕ entails (resp., disentails) the OSF constraint $\exists U.\, (U \doteq X \,\&\, \psi)$ if and only if the normal form ψ' of $\psi \,\&\, U \doteq X$ relatively to ϕ is a conjunction of equations making up a functional binding (resp., is the* false *constraint $\psi' = \bot$).*

4 Conclusion

We have overviewed a complete and correct system for deciding entailment and disentailment of constraints over order-sorted feature strcutures. One motivation for this system is parameter-passing for functions in LIFE, but it is general and relevant to all concurrent constraint languages. We used a technique of relative simplification [4, 7] which amounts to normalizing a constraint in the context of another. This yields an incremental system. Let us mention here that we can also prove the independence property of negative constraints.

Further work extending this should be to generalize our scheme to so-called deep guards over OSF structures whereby guards are not limited to plain OSF constraints but may also contain relational atoms defined by clauses. This is particularly relevant to LIFE in order to explain matching over objects with attached relational constraints. This study in currently under way and will be reported soon.

References

[1] Hassan Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351 (1986).

[2] Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215 (1986).

[3] Hassan Aït-Kaci and Roger Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89 (1989).

[4] Hassan Aït-Kaci and Andreas Podelski. Functions as passive constraints in LIFE. PRL Research Report 13, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (June 1991). (Revised, November 1992).

[5] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. PRL Research Report 11, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1991). (Revised, October 1992; to appear in the *Journal of Logic Programming*).

[6] Hassan Aï-Kaci and Andreas Podelski. Entailment and disentailment of order-sorted feature constraints. In Andrei Voronkov, editor, *Proceedings of the Fourth International Conference on Logic Programming and Automated Reasoning*. Springer-Verlag (1993, to appear).

[7] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. In *Proceedings of the 5th International Conference on Fifth Generation Computer Systems*, pages 1012–1022, Tokyo, Japan (June 1992). ICOT. (Full paper to appear in *Theoretical Computer Science*).

[8] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In David H. D. Warren and Peter Szeredi, editors, *Logic Programming, Proceedings of the 7th International Conference*, pages 31–46, Cambridge, MA (1990). MIT Press.

[9] Michael Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876, Cambridge, MA (1987). MIT Press.

[10] Vijay Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM (January 1990).

Separability of Polyhedra and a New Approach to Spatial Storage *(Extended Abstract)*

Alexander Brodsky Catherine Lassez

I.B.M. Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Efficient storage and access methods for large amounts of spatial objects are key issues in Geographic Information Systems (GIS), Computer Aided Design (CAD), VLSI design and also Linear Constraint Databases (LCDBs) [BJM92], a new application domain in which objects are convex multidimensional polyhedra represented as conjunctions of linear constraints over real variables. Typically, the first step of query processing is the filtering out of irrelevant information.

We propose a new filtering method which is based on pre-evaluation of projections of objects (polyhedra) on a number of selected axes. We are concerned with how to achieve any desired quality of filtering by selecting (a minimum number of) optimal axes, while keeping storage overhead low.

Filtering in Spatial Queries

Typical spatial queries deal with intersection and containment of objects. For example, given a particular object (the query object) we may ask what are the objects in the database that intersect it or contain it or are enclosed in it. We may also ask what are the pairs of objects that intersect each other. To answer this type of queries the idea of *filtering* using *minimum bounding boxes* (MBB), also called *minimum bounding rectangles*, is widely used.

In the case of GIS (two-dimensional case), the MBB of a given object is the smallest rectangle that encloses the object and whose edges are parallel to the standard coordinate axes. The MBB's are stored as pairs of intervals along with the objects in an efficient access structure such as R tree, R^+ tree, R^* tree, or a structure based on combination interval, segment and range trees [SiW82, Ed83].

The evaluation of a query consists of a *filtering* and a *refinement* steps. In the filtering step, the access structure is used to retrieve only relevant objects, that

is, those objects whose MBB's intersect the MBB of the query object. In the refinement step, each of the retrieved objects is tested for intersection with the query object. For instance, in Figure 1, the filtering step in retrieving the objects that intersect O_4 gives O_3 and O_5. The refinement step eliminates O_3 that does not intersect the query object.

The filtering method based on MBB's is simple and has a number of major advantages. First, it is economical in storage and access, since only two intervals are stored in addition to each object. Second, because the access structure manages only pairs of intervals, and not the objects themselves, there is a clear separation between the complexity of the object geometry and the complexity of the search (access methods) [Ni90]. Third, efficient access methods for rectangles (pairs of intervals) such as R trees [Gut84, RL84, RL85], R^+ trees [SRF87], R^* trees [BKSS90], or based on combination of interval and range trees [Ed83, SiW82] have been developed. The latter structure, for example, has a worst case time bound of $O(log^2 n + k)$ for search, where n is the total number of rectangles and k is the number of rectangles that intersect the query rectangle (the space requirement, however, is $O(nlogn)$). Whereas no worst case bounds can be guaranteed by existing spatial access methods that work directly on objects.

The major drawback of filtering with MBB's is that it might be ineffective when many disjoint objects have intersecting MBB's. For instance, in Figure 2, whatever the query object, the filtering provides no help as all MBB's intersect.

There are alternative methods to MBB's, based on decomposition of space into disjoint cells. These include uniform grid method [Fr84], quadtree-based methods [Ta82, SaW85, NS87, Or89], $R+$ and $R*$ trees applied to objects, and cell tree [Gun87]. While these methods reduce the problem of poor filtering, they operate on a number of cells usually far larger than the number of objects themselves.[1]

For the multidimensional case in LCDB the use of disjoint decomposition of space is unfeasible bacause the space might be of dimension of hundreds more. For the same reason the MBB's method is not applicable as is, but can be generalized to deal only with a manageable number of axes. Still, the quality of filtering might be poor.

A New Approach

We propose a new approach to achieve any desired quality of filtering by generalizing the concept of MBB's, while preserving the advantages described above. We enclose each object in a *Minimum Bounding Polybox* (MBP), defined as the minimum polyhedron that encloses the object and whose facets are normal to preselected axes. These axes are not necessarily the standard coordinate ones and furthermore their number is not determined by the dimension of the space. The

[1] In grid and quadtree methods there is a trade off between the resolution of the cells (and thus quantity of the cells) and the effectiveness of filtering.

idea is to select a minimal number of optimal axes that maximize the quality of filtering while keeping storage overhead low. For example, for the objects in Figure 2, only one axis, X', is sufficient to obtain optimal filtering as is shown in Figure 3, since all MBPs are disjoint. Note that the MBPs here are unbounded. It is easy to verify that two MBP's are disjoint if and only if their projections on at least one axis are disjoint.

It is now assumed that the objects considered are convex polyhedra (more complex objects can be approximated as unions of convex polyhedra). We address the problem of minimizing the number of axes required to achieve a given quality of filtering as well as the reverse problem of optimizing the quality of filtering when the number of axes is given. We say that an axis separates two objects if their projections on this axis are disjoint. We also say that a set of axes separates a set of pairs of objects if each pair is separated by at least one axis. In the full paper we define formally the *quality of filtering* for a given collection of objects and axes. For our purpose here, it is sufficient to state that there is a $1-1$ correspondance between the quality of filtering and the number of pairs of objects separated by the axes.

We prove that the following is computable for a given set of objects:

1. The minimum number L of axes needed to separate all N pairs of disjoint objects.

2. The maximum number of pairs of objects that can be separated using l axes, for $1 \leq l \leq L$.

3. A collection of l axes separating the maximum number of pairs as above.

4. The minimum number l of axes needed to separate K pairs of disjoint objects, for $0 \leq K \leq N$.

5. A collection of l axes separating any K pairs of objects as above.

In order to prove the above we introduce the concept of *separability classification*. Given a collection of objects $O_1, \ldots, O_n$, we say that two axes are *equivalent* if they each separate the same set of pairs of objects. We define the *separability classification* of $O_1, \ldots, O_n$ as the set

$$\{(E_i, S_i)\}_{i \in I}$$

where each E_i is a finite representation of an equivalence class of axes and S_i is a maximal subset of pairs of objects separated by E_i, and I is an index set of all equivalent classes of axes, excluding the class that separates no pairs and the empty class. Moreover, the finite representation of E_i is required to satisfy the property that membership of an axis in E_i is decidable and a representative of E_i is computable.

Note that each equivalence class may contain an infinite number of axes, and thus the existance of its finite representation with the required properties is not clear. Also, since the equivalence classes of axes can be represented in many different ways, we may have many separability classifications for the same set of objects. It is clear, however, that the separability classification, if it exists, is unique up to representation of equivalence classes. An *instance* of a separability classification is a collection of axes, one from each equivalent class E_i.

Theorem 0.1 *A separability classification of objects O_1, ..., O_n exists and is computable.*

The proof of this theorem, to be found in the full paper, is constructive and thus provides an algorithm to actually compute a separability classification. It should be noted that the concept of separability classification is a general tool for many potential applications, e.g. within the framework of computational geometry.

¿From a computational point of view, this algorithm is exponential for two reasons. First it requires to consider all subsets of pairs of objects. Second, the test for each subset requires solving an exponential number of linear programs. Moreover, just computing a representative of an equivalence class also requires an exponential number of linear programs. Even for the two-dimensional case this algorithm remains exponential. However, we show that:

Theorem 0.2 *Given a set of two-dimensional objects O_1, ..., O_n there exists a separability classification of at most $O(n^2)$ pairs (E_i, S_i). Furthermore, both testing instance membership and evaluating an instance of the classification takes $O(n^2)$ time. Moreover, the separability classification can be computed in polynomial time in the total number of constraints (or alternatively extreme points) used to represent O_1, ..., O_n.*

Acknowledgment:
The authors wish to thank for Jean-Louis Lassez for his help.

References

[BKSS90] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeder, The R^*-tree: An efficient and robust access method for points and rectangles, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 322-331, Atlantic City, May 1990.

[BJM92] A. Brodsky, J. Jaffar, M.J. Maher, Toward Practical Constraint Databases, *IBM Research Report*, IBM T.J. Watson Research Center, 1992.

[Ed83] H. Edelsbrunner, A new approach to rectangle intersections, Part II, *International Journal of Computer Mathematics*, 13, pp. 221–229, 1983.

[Fr84] W.R. Franklin, Adaptive grids for geometric operations, *Cartographica 21, 2 & 3*, pp. 160-167, 1984.

[Gun87] O. Gunther, Efficient structures for geometric data management, *Lecture Notes in Computer Science 337*, Springer Verlag, Berlin, 1988.

[Gut84] A. Guttman, *R*-trees: A dynamic index structure for spatial searching, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Boston, MA, pp. 47-57, 1984.

[Ni90] J. Nievergelt, 7 ± 2 Criteria for Assesing and Comparing Spatial Databases, *Symp. on the Design and Implementation of Large Spatial Databases*, pp.89-114, New-York, Springer-Verlag.

[NS87] R.C. Nelson, H. Samet, A population analysis for hierarchical data structures, *Proc. of the SIGMOD Conf. San Francisco*, May 1987, pp. 270-277.

[Or89] J.A. Orenstein, Redundancy in spatial databases, *Proc. of the SIGMOD Conf., Portland, OR*, June 1989, pp. 294-305.

[RL84] N. Roussopoulos, D. Leifker, An introduction to PSQL: A pictorial structured query language, *IEEE Workshop on Visual Language*, Hiroshima, Japan, pp. 77-84, 1984.

[RL85] N. Roussopoulos, D. Leifker, Direct spatial search on pictorial databases using packed R-trees, *Proc. ACM SIGMOD*, pp. 17–31, 1985.

[SaW85] H. Samet, R.E. Webber, Storing a collection of polygons using quadtrees, *ACM Trans. on Graphics 4*, 3, pp. 182-222, July 1985.

[SiW82] H.W. Six, D. Wood, Counting and reporting intersections of *d*-ranges, *IEEE Trans. Computing C-31*, pp. 181–187, 1982.

[SRF87] T. Sellis, N. Roussopoulos, C. Faloutsus, The R^+-tree: A dynamic index for multidimensional objects, *Proc. 13th Int. Conf. Very Large Data Bases*, pp. 507–518, 1987.

[Ta82] M. Tamminen, Efficient spatial access to a data base, *Acta Polytechnica Scandinavica*, Mathematics and Computer Science Series No. 34, Helsinki, Finland, 1981.

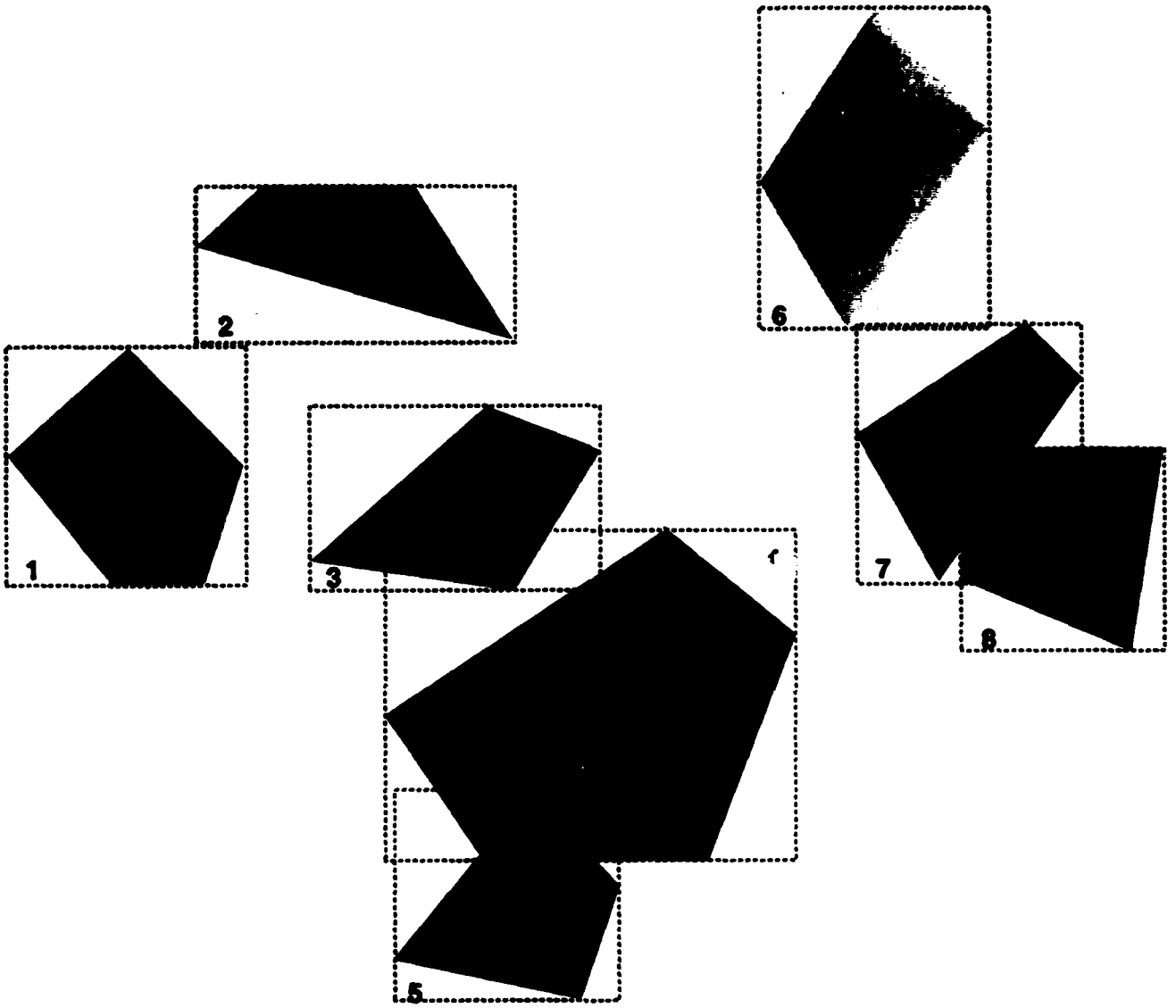


Figure 1.

LASSEZ at YKTVMH - FIG1 PSSN processed on 01/29/93 10:42:41 - Dist

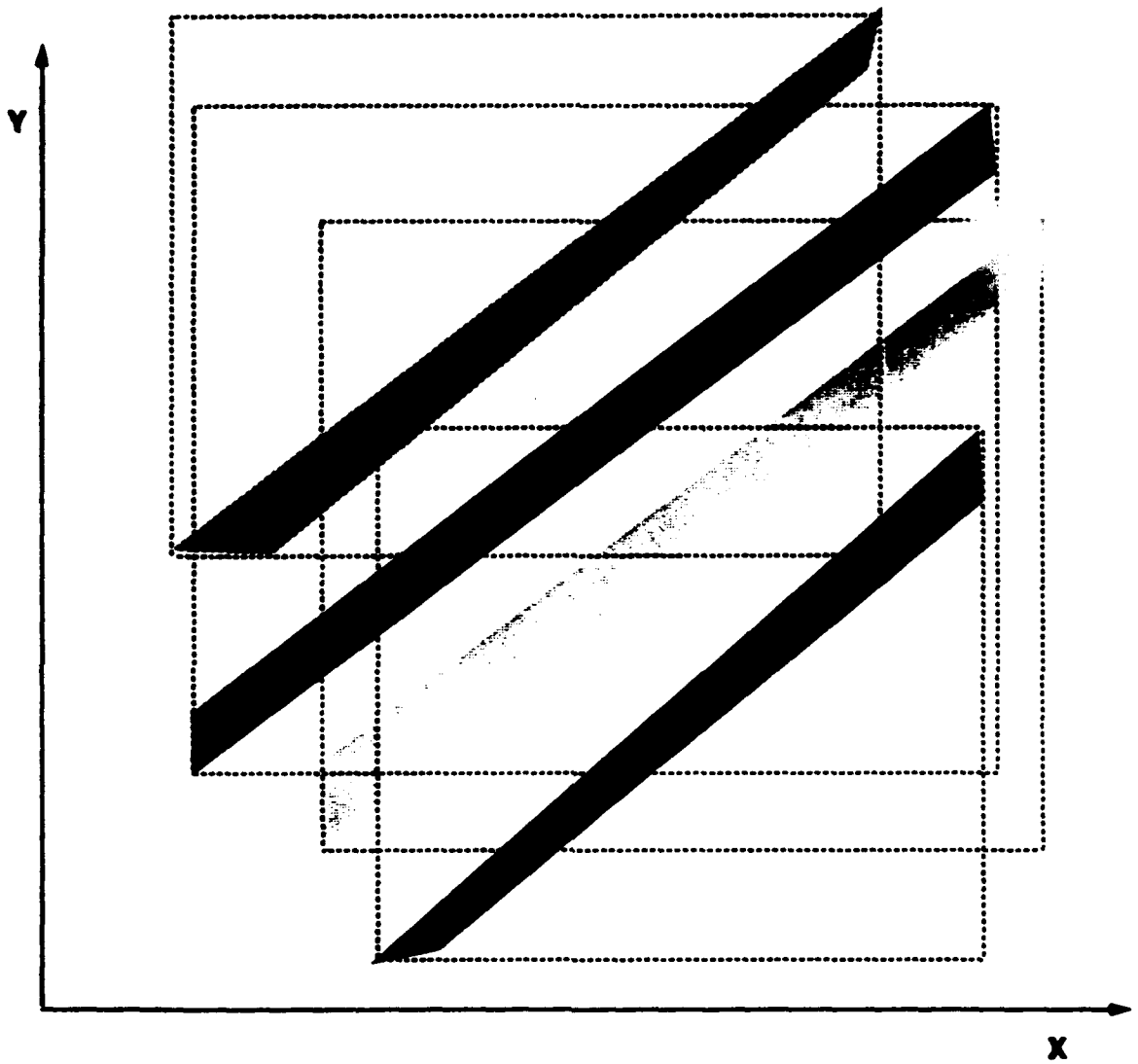


Figure 2.

LASSEZ at YKTVMH - FIG2 PSBN processed on 01/26/93 10:37:47 - Dist

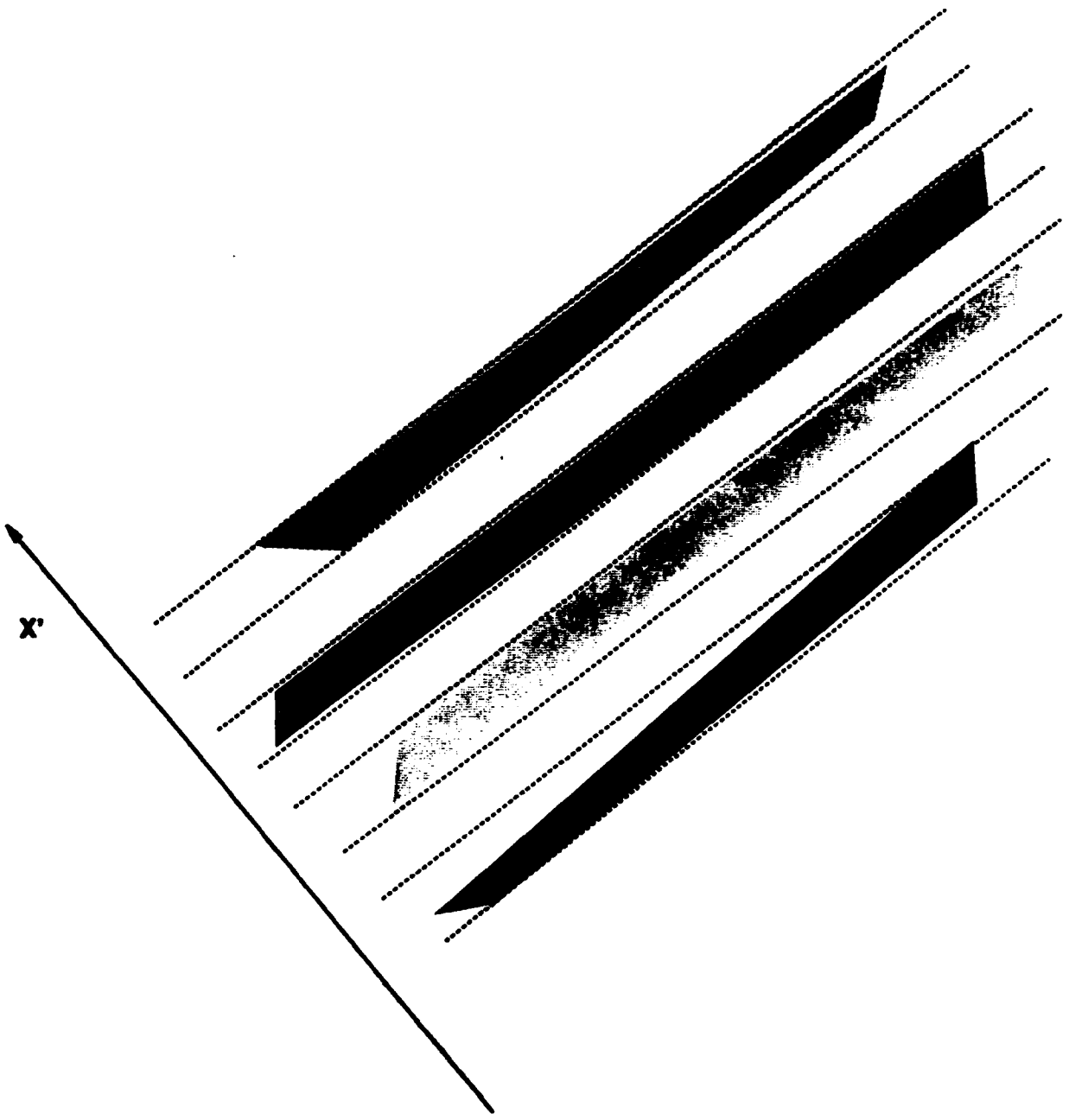


Figure 3.

LASSEZ at YKTVMH - PID3 PSBIN processed on 01/20/93 10:36:37 - Dist

Constraint Optimization using Preference Logics: A New Role for Modal Logic

Allen L. Brown, Jr. Surya Mantha Toshiro Wakayama
Webster Research Center
Xerox Corporation

April 4, 1993

Abstract

A family of modal logics of preference is presented. It is argued that modeling preference as a modal operator captures it at the correct level of granularity and extracts the logical core of the notion of preference. Next, the problem that motivated the development of preference logics is discussed in some detail. The paper ends with a list of other areas in which preference logics have shown promise.

1 Introduction

The purpose of this paper is to outline the development of a logical theory of preference. While the work was motivated by an extremely concrete problem, the resulting theory turns out to be surprisingly general and powerful in its scope, applicability and expressive power. In the first half of this paper, we introduce a family of modal logics of preference followed by a very cursory look at the historical approaches to preference in the literature of decision theory, philosophical logic and artificial intelligence. Following that, we describe – perforce briefly – the problem that motivated this line of research. In ending we list some of the other areas in which preference logics have found application.

2 Modal Logics of Preference

The research reported in this paper can be viewed as the design of a logical language for stating symbolic optimization problems succinctly. Informally, any optimization problem consists of a set S of constraints that define the solution space, and an *objective function* O that identifies one or more of the solutions as *optimal solutions* to S. The task at hand is to device a language – with a precise model-theoretic semantics – in which (executable) specifications of O can be given.

2.1 Logic of Preference and Prohibition $\mathcal{P}_2$

Syntax : We add to the language $\mathcal{L}$ of propositional logic two new monadic modal operators P_f and P_b. The rules of formation of $\mathcal{L}_{P_2}$ include all the rules of $\mathcal{L}$ in addition to

- If F is a formula then $P_f F$ is a formula, and $P_b F$ is a formula.

Semantics : A *preference frame* $\mathcal{M}$ is an ordered pair of the form $\langle \mathcal{W}, \preceq \rangle$ where $\mathcal{W}$ is a set of possible worlds and $\preceq$ a binary (*preference*) relation on them. A *preference model* is a preference frame along with a valuation function $\mathcal{V}$ that determines the truth of atomic formulae at individual worlds. Thus,

$$\models^{w}_{\mathcal{M}} F \text{ iff } \mathcal{V}(F, w) = true.$$

for all atomic formulae F. Assuming the standard semantics of propositional connectives, the semantics of P_f and P_b are given by:

$$\models^{w}_{\mathcal{M}} P_f F \text{ iff } \forall v \in \mathcal{W} \models^{v}_{\mathcal{M}} F \text{ implies } w \preceq v$$
$$\models^{w}_{\mathcal{M}} P_b F \text{ iff } \forall v \in \mathcal{W} \models^{v}_{\mathcal{M}} F \text{ implies } w \not\preceq v$$

Roughly, the semantics of $P_f F$ captures the intuition that F suffices for preference. The formula F is sufficient to induce a preference ordering between two worlds. *F is a preference criterion at the world w.* v is at least as good as w if $w \preceq v$. P_b, on the other hand, precludes such a relationship.

Axiomatics : We assume that $\mathcal{P}$ is equipped with all the axiom schemes and rules of standard propositional logic. In addition, we have the following rules.

PI if $\vdash (A_1 \wedge \cdots \wedge A_n) \rightarrow A$ then $\vdash (P_f \neg A_1 \wedge \cdots \wedge P_f \neg A_n) \rightarrow P_f \neg A$ for $n \geq 0$

PBI if $\vdash (A_1 \wedge \cdots \wedge A_n) \rightarrow A$ then $\vdash (P_b \neg A_1 \wedge \cdots \wedge P_b \neg A_n) \rightarrow P_b \neg A$ for $n \geq 0$

We now define two derived operators A_m (*admissibility*) and D_m (*dismissibility*).

- $A_m F \equiv_{def} \neg P_b F$, and
- $D_m F \equiv_{def} \neg P_f F$.

Semantically

- $\models^{w}_{\mathcal{M}} A_m F$ iff $\exists v \in \mathcal{W} \models^{v}_{\mathcal{M}} F \wedge w \preceq v$,
- $\models^{w}_{\mathcal{M}} D_m F$ iff $\exists v \in \mathcal{W} \models^{v}_{\mathcal{M}} F \wedge w \not\preceq v$.

Thus, we have a spectrum of *indifferences*, ranging from *admissibles* on the one (weaker) end to *dismissibles* on the other end. The logic $\mathcal{P}_2$ is equivalent to a bimodal logic that characterizes complementary bimodal frames. In order to have completeness with respect to the class of standard preference models, we need to strengthen our axiomatization with what we shall call the Humberstone schema [17] (who was among the first to study the complement of the usual world relation). In our framework, if Π_1 and Π_2 stand for strings of any (including zero) length of occurrences of the weak operators A_m and D_m, then

PH : $\Pi_1(P_f \alpha \wedge P_b \beta) \rightarrow \neg \Pi_2(\alpha \wedge \beta)$

The schema **PH** is valid in all preference frames. In fact, its presence is required to show the completeness of $\mathcal{P}_2$.

Theorem 2.1 *The logic of preference $\mathcal{P}_2$ (including the schema **PH**) is sound and complete with respect to the class of standard preference models.*

2.2 Logic of Feasible Preference $\mathcal{P}_1$

$\mathcal{P}_1$ too, is a modal logic of two relations that *interact* with each other. The motivation for this interaction is to capture the intuition that in order to get to the *optimal* (or best) world, one needs to be able to talk about worlds that are feasible from the standpoint of the current world. The motivation underlying this whole enterprise is to devise a formal language and logic in which optimization problems can be stated precisely. Thus, if w_2 is feasible from w_1, and $w_1 \preceq w_2$ and $w_2 \not\preceq w_1$, then it is possible to move from the solution w_1 to w_2. If, however, w_2 were not feasible from w_1, then it would not be possible to move from w_1 to w_2 even if w_2 were preferred to w_1. This interaction between the two relations is fundamental to modeling any situation that is of computational interest: in particular, it is crucial to any *search* based computation.

Syntax : We add to the language $\mathcal{L}_m$ of a normal modal logic – equipped with the modal operators $\Box$ and $\Diamond$ – the modal operator introduced above, i.e., P_f and its associated formation rule.

Semantics : A $\mathcal{P}_1$ *preference frame* $\mathcal{M}$ is a triple of the form $\langle \mathcal{W}, \mathcal{R}, \preceq \rangle$ where $\mathcal{W}$ and $\mathcal{R}$ are as in standard Kripke frames and $\preceq$ is a binary relation over $\mathcal{W} \times \mathcal{W}$ which is a subset of $\mathcal{R}$. A $\mathcal{P}_1$ *preference model* is a $\mathcal{P}_1$ preference frame with a valuation function $\mathcal{V}$ that determines the truth of atomic formulae at individual worlds. Assuming the usual valuation of formulae at possible worlds, the semantics of the modal operators are given by:

- $\models^w_{\mathcal{M}} \Box F$ iff $\forall v \in \mathcal{W}\ w \mathcal{R} v \ \rightarrow \models^v_{\mathcal{M}} F$
- $\models^w_{\mathcal{M}} P_f F$ iff $\forall v \in \mathcal{W} \models^v_{\mathcal{M}} F \wedge w \mathcal{R} v \rightarrow w \preceq v$

Axiomatics : $\mathcal{P}_1$ is equipped with all the axioms and rules of $\mathcal{P}$ and the normal modal logic $\mathcal{K}$, and has the following axioms.

PPS : $\vdash \Box\neg A \rightarrow P_f A$

PIR : $\vdash P_f A \rightarrow \neg A$

T : $\vdash \Box A \rightarrow A$

PIR is valid in the class of preference frames with an *irreflexive* preference relation. **T** is valid in the class of preference frames with a *reflexive* feasibility relation. **PIR** and **T** are needed to show the completeness of $\mathcal{P}_1$.

The language of $\mathcal{P}_1$ is rich enough to allow us to express general preference principles. Because our objective is to characterize the intuitive notion of *better*, we are interested in syntactically characterizing *irreflexivity*, *transitivity* and *asymmetry*. Irreflexivity of the preference relation is characterized by **PIR** above. In the case of *transitivity* and *asymmetry*, **PTR** and **PAS** below ensure these properties.

PTR : $(P_f A \wedge \Diamond(P_f B \wedge A)) \rightarrow P_f B$

PAS : $\neg\Diamond((P_f A \wedge B) \wedge \Diamond(A \wedge P_f B))$

PAS also *expresses* asymmetry. A formula is said to *express* a class of frames, if and only if it is valid in all and only the frames in that class. **PTR** only ensures transitivity in all *supported preferential models*.

Definition 2.1 *A preferential model is said to be* supported, *iff, if for any two worlds w and v if $w \preceq v$, then there exists a formula $P_f A$ such that $w \models P_f A$ and $v \models A$.*

Supported preferential models will be important in the treatment of preferential theories and their intended preference models [8], [21]. There are, however, transitive preference frames where it is not valid. To characterize transitivity exactly, the notion of *prohibition* introduced in $\mathcal{P}_2$ above is needed.

3 Granularity of Preference

The notion of preference is fundamental to computing. From combinatorial optimization to the minimal models of logic programs and circumscriptive theories, it plays a fundamental role in computer science. Preference has been used directly and/or indirectly to represent a reasoning agent's knowledge in a variety of computational contexts in artificial intelligence. The various proposals for doing nonmonotonic reasoning in logic can be viewed as a form of preferential reasoning, where the agent bases his/her beliefs on one or more of the preferred models of an underlying logical theory (which encodes the agent's partial information about his/her environment), and the preferred models are given by the agent's biases regarding *completing* this partial information. Circumscription in its various flavors [19], can be viewed as a mechanism – though somewhat cumbersome – of programming *preference* via the more special notion of *minimality*. Even such syntactic formalisms as Reiter's default logic [22] and Gabbay and Makinson's cumulative inference relations [20] have been given semantics based on preference orderings on models. Shoham [23], in his doctoral dissertation, proposed a general semantic model of nonmonotonic reasoning based on a preference ordering on the models of a theory. The preference relation in his work was *implicit* and there was no syntactic way of manipulating it. It is somewhat puzzling that though preference has been the representational and computational mechanism at the core of all nonmonotonic reasoning systems, very little attention has been paid to modeling it directly in the syntax. Researchers have spent much more time and effort formalizing other intensional notions such as *belief* and *knowledge* (and with mixed results at best).

Decision theory, economics, ethics and philosophical logic are other disciplines where the notion of preference has been studied extensively. But in almost all accounts, [18], [14], [24], [25], [13], [15], preference is taken to be a special binary relation on individual propositions, i.e., those objects that can be represented by sentences of the underlying logical language (these propositions are supposed to characterize *states of affairs*). The preference statement pPq is intended to mean that p is preferred to q. Competing – and mutually inconsistent – theories of preference have been proposed over the decades. Much of the controversy and debate has been around the question: *what are the logical properties of preference?* Is preference asymmetric [1]? Much has been written for and against transitivity [16], [11]. The problem in all these attempts was that the scope of preference was extremely *local*, i.e., over individual propositions. Having committed to such a fine level of granularity, one has to make rather strong commitments as to what *logical* properties the preference relation enjoys under all circumstances, i.e., is it asymmetric, transitive and so forth.

The use of preference in nonmonotonicity, of course, lost this local aspect altogether and saw a drastic shift to the extremely global, with disastrous implications for computational efficiency. Thus in circumscription, we talk about truth in **all** minimal models, default logic has the global notion of an extension (albeit a purely syntactic one) and Gabbay's nonmonotonic inference relation $A \mid\!\sim B$ is informally understood as *B is true in all the most preferred models of A.*

Our own formalization of preference using modal logic was not an accident. In a later section, we sketch in some detail the problem of specifying style specifications in the layout of a structured document. Intuitively, the problem came down to one of imposing preference orderings among a set S of collections of propositions. Each collection of propositions C_i describes the properties of a

laid out document and the different collections in a set S are different ways of laying out the same document. In the case of documents the collections are finitary (documents being finite entities), but any general theory of preference should be able to handle infinitary objects as well.

An immediate consequence of this is that it becomes difficult to model preference as a binary relation P (as is commonplace in the decision theoretic literature) because it is not convenient to talk explicitly about the referents of this relation. But the notion of a *preference criterion* arises naturally. For instance, in laying out a document, one might prefer a layout with fewer number of pages to one with more pages. If *pagenos* is a *property* of a layout, then it is also a preference criterion. Because what we want to talk about *explicitly* in our logic are preference criteria, that make one *state of affairs* (possibly infinite collection of propositions) preferable to another state of affairs, modal logic is a natural choice. The modal account also captures another key property of preference, i.e., *locality*. Preference is local in nature. This locality exists along, at least, two dimensions. On the one hand, we speak of preferences between objects without being aware of them in their totality. On the other hand, we often speak of relative preferences between objects without being consciously aware of a *maximum* or *maximal* object. In fact, it is this second aspect of locality that nonmonotonic logics and circumscriptive theories have sacrificed. Modal preference logics also allow us to have preferences about our preferences (see [18] for a lively debate on this).

If the preference operator P_f looks very similar to the alethic operator $\Box$, that is because it is. Simply put, it is $\Box$ on the complement of the usual world relation. $\Box$ itself wouldn't have served our purpose. Keep in mind that we want to impose orders over possible situations (that are alternative to one another) based on very local information about these situations. The weakest normal modal logic K has the rules $\frac{A}{\Box A}$ $\frac{\neg A}{\neg \Diamond A}$. This makes it difficult to give an interpretation to $\Box$ and $\Diamond$, that could correspond to the notion of preference: *i.e.*, an interpretation that would have the truth of $\Box p$ at a given world mean that all worlds at which p is true are at least as good as the current world. The models of any non-trivial (*i.e.* non-empty) system would have a universal preference relation since the above inference rule would make all the theorems of a theory, preference criteria, something, clearly not desirable.

4 The Backdrop: Declarative Document Description

The particular problem that motivated this work was that of designing a formal language that would facilitate precise and unambiguous specification of layout directives and style information in structured documents. This itself was part of a larger project which involved the design and development of a formal, mathematical model of document processing. It can be argued that typical document specification languages (which we shall call *markups*), together with the programs that interpret them, constitute computational theories of documents. The problem with such markups is that the only explicit semantics provided is that of the accompanying interpreters. The document specification language that we have in mind should have a declarative model-theoretic semantics that should serve as the basis of any computational interpretation of programs in that language.

Inspite of their numerous drawbacks, both SGML (Standard Generalized Markup Language) [12] and ODA (Office Document Architecture) – ISO standards for document interchange have the following attractive features.

- There is a clear separation of *structure* from *content*. SGML uses the grammatical paradigm and grammars (with some very rudimentary attribution mechanism) are used to describe classes of structured documents, i.e., one can, in SGML, write different grammar-like entities to describe the generic structures of books, technical articles, forms and so forth. An instance

of a document (belonging to one of the above classes) would be an attributed parse tree of the corresponding grammar. ODA uses object-oriented terminology and principles to do essentially the same.

- The notion of a primary *logical structure* of a document that has no mention of any processing information (such as layout, for instance). In fact, SGML does not concern itself with any processing (interchange, layout, recognition to name a few). The ODA standard deals with the logical as well as the *layout structure* of documents.

We give below an example of a (generic) logical and (generic) layout structure using regular right hand part grammars(i.e. context free grammars with regular expressions on the right hand sides).

Logical Structure	**Layout Structure**
$LogicalRoot \rightarrow Section^+$	$LayoutRoot \rightarrow Page^+$
$Section \rightarrow Title\ Paragraph^+$	$Page \rightarrow Line^+$
$Paragraph \rightarrow Word^+$	$Line \rightarrow Word^+$

Thus in ODA, the complete description of a document is given by the following : a generic logical structure (roughly a grammar), a specific logical structure (roughly a parse tree), a generic layout structure and a specific layout structure. The correspondence between logical structures and layout structures is made through special attributes (of the logical structures) that link logical categories with layout categories. These constitute what is called *style specification*.

The above outline is indeed very rough, but we hope it sets the stage for the issue that we shall discuss next. In the spirit of ODA, assume that a document is described by the quadruple $\mathcal{D} = \langle G_{logical}, G_{layout}, Coord, t_{logical} \rangle$, where $G_{logical}$ and G_{layout} specify the generic logical and layout structures respectively. $t_{logical}$ is the parse tree, i.e., marked up content according to $G_{logical}$, and *Coord* describes the *correspondences* between the generic logical and generic layout structures. For instance, *Coord* for the above pair of grammars could be given by the following productions

Coordination
$LogicalRoot \rightarrow LayoutRoot$
$Section \rightarrow Page^+$
$Paragraph \rightarrow Line^+$

Such coordination grammars relate logical entities with layout entities. More abstractly, *coordination* can be viewed as a *mapping* from one class of structured objects (attributed trees) to another such class. Grammars are one concrete way of specifying such mappings. Thus, a section will be laid out in a sequence of pages and a paragraph will be laid out as a sequence of lines. Given a $\mathcal{D}$, the problem of laying out a document can be formulated as the problem of computing a t_{layout} which belongs to the language of G_{layout} and is coordinated with respect to $t_{logical}$ according to the coordination *Coord*. The notion of *coordinated with* has a precise mathematical meaning which shall, however, not concern us in the rest of this paper. In general, any document processing problem can be formulated in the way given above and we refer the reader to [5] for a detailed description of the formalism.

Line Breaking G_{lb}
$Paragraph \rightarrow Line^+$
$Cost(Paragraph, C) :- bag_of(X, Cost(Line, X), A), Sum(A, C)$
$Line \rightarrow Word^+$
$Cost(Line, F(Word^+)$

Let us now motivate the central concern of this paper, i.e., preference. Consider the above grammar for line breaking. The first production is taken from our example coordination and the second production is taken from our example layout grammar. We have also attached above, *Cost* attributes with the nonterminals *Paragraph* and *Line*. These attributes are specified by Horn clause logic programs. F is a function that given the words in a line computes the cost (or

badness) of that line. The syntax above is specified somewhat informally, but should get the point across. (The reader is referred to [5] for details). Those familiar with the Knuth-Plass line-breaking algorithm (which is at the heart of TEX) will immediately see the parallels. The grammar G_{lb} above, is thus, the grammaticatical specification of the well-known line-breaking problem.

The key observation that we now make is that G_{lb} is highly ambiguous. Given a paragraph (i.e., a sequence of words) there are (exponentially) many ways of breaking it into lines. In other words, given a string of words s, there are exponentially many ways $\langle t_i, \pi_i \rangle$ (where t_i is a tree and π_i is its associated logic program for its attributes), in which it can be parsed by G_{lb}. The Knuth-Plass algorithm computes the best line-break by selecting the one with the lowest cost. The monotonicity of the cost function allows the use of dynamic programming and gives a linear time algorithm.

Consider, however, our grammatical formulation G_{lb}. As it stands, there is nothing in the specification that states that we prefer the parse tree with the lowest value of *Cost*. We would like a declarative statement of our preference for the parse tree with the lowest value of *Cost*. The range of this comparison (for a given paragraph p) is over all pairs $\langle t_i, \pi_i \rangle$ that are admissible line-breaks for p with respect to the grammar G_{lb}. Identifying the parse trees with their associated logic programs, we finally end up with the following semi-formal statement of the problem: *the design of a language that will allow the placement of preference orders on collections of propositions (in our case, the attributes of the parse trees).*

Specifications in such a language would correspond very closely to the way actual graphic designers and layout specialists work. Rather than tell a layout system *how to* perform the layout, designers prefer to make declarative assertions regarding what kind of layout they would like, i.e., *I would like a layout with the fewest number of pages*, or equivalently stated, *Between two layouts of the same material, I would prefer the one with fewer pages.*

5 Applications: Current and Future

Using $\mathcal{P}_1$ we were able to specify the preference criteria required to imposed preference orders on the various competing layouts of a structured document. We have extended the attribute grammar formalism to incorporate *ambiguity* and *preference*, giving rise to what we call *Preferential Attribute Grammar Schemes* (PAGS). The details can be found in [5]. Work is in progress on implementing a WYSIWYG editor with PAGS as the underlying representation for structured documents. We then set about studying the applicability of the logics to other problem domains. We enumerate below some of the areas in which preference logics have already shown immense promise.

Deontic Logic Consider the following definitions of the deontic modalities Obligation ($\bigcirc$) and Permission (P) [4] [6].

OD2 $\bigcirc p \equiv_{def} P_f p \wedge A_m p$

P1 $PA \equiv_{def} \neg \bigcirc \neg A$.

P2 $PA \equiv_{def} D_m \neg A \wedge A_m A$.

P3 $PA \equiv_{def} A_m A$.

The definition **OD2** steers clear of all the paradoxes of standard deontic logic [10], [2]. The three ways of defining permission give rise to different notions of permission, ranging from the so called *free-choice permission* to weak permission (as in **P3**). We are confident that preference logics will have significant impact in legal reasoning and the formal specification of normative systems.

Nonmonotonic Reasoning In [8], we defined the notion of a preferential theory and recast nonmonotonicity as computing the optimal worlds in the intended model of a preferential theory. Default and/or uncertain knowledge of the agent are coded as preference criteria. For instance, the simple default *birds normally fly* is coded as the formula, $Bird(X) \wedge \neg Flies(X) \rightarrow P_f(Bird(X) \wedge Flies(X))$. Thus, those models where a given bird flies are preferred over those where it does not.

Logic Programming Using the notion of a preferential theory, we were able to give a finitary characterization of the stable models of a normal logic program [7].

Constraint Relaxation In a series of papers [3], [26], Borning and his students have introduced the notion of constraint hierarchies in logic programming. This work was done in the context of Constraint Logic Progamming, where a partial order (giving the order in which to relax the constraints if all of them cannot be satisfied) is placed on the constraints on the right hand side of a clause. Using preference logics, we have generalized this to, what we call, *Relaxable Horn Clauses* [9]. In RHC we place partial orders on the bodies of definite clauses. These partial orders are interpreted as a specification of *relaxation criteria* in the proof of the consequent of a relaxable clause, i.e., the order in which to relax the conditions of truthhood of the consequent if all the goals in the body cannot be satisfied. Preference logics enable us to characterize these preference orders, both syntactically and semantically.

We believe that preference logics provide a unifying framework for the use of modal logic in ambiguous computational contexts, be they in structural descriptions using attribute grammars, nonmonotonic reasoning, or normative specifications of an ethical agent's obligations. Some of the areas where the use of preference logics to represent domain knowledge would be immensely profitable are *abduction, model based diagnostic reasoning* and *planning*. Applications in other areas such as decision theory, risk management also seem promising. We have already begun investigating these and other issues that are of a more algorithmic nature. We hope that interaction with other researchers at the workshop will facilitate in bringing out interesting problems and more application areas.

6 Acknowledgements

We would like to thank Profs. Anil Nerode (Cornell) and Howard Blair (Syracuse) for their encouragement and involvement in various stages of this two year investigation.

References

[1] ACKERMANN, R. Comments on n. rescher's semantic foundation for the logic of preference. In *The Logic of Decision and Action*. 1967.

[2] AQVIST, L. Deontic logic. In *Handbook of Philosophical Logic*, D. Gabbay and F. Guenthner, Eds. D. Reidel Publishing Company, Dordrecht, 1984, pp. 605–714.

[3] BORNING, A., AND ET. AL., M. M. Constraint hierarchies and logic programming. In *Sixth International Conference on Logic Programming* (June 1989), pp. 149–164.

[4] BROWN JR., A. L., MANTHA, S., AND WAKAYAMA, T. Preferences as normative knowledge: Towards declarative obligations. In *First International Workshop on Deontic Logic in Com-*

puter Science (Amsterdam, The Netherlands, 1991), J. J. C. Meyer and R. J. Wieringa, Eds., pp. 142–164.

[5] BROWN JR., A. L., MANTHA, S., AND WAKAYAMA, T. The declarative semantics of document processing. In *PODP: Principles of Document Processing, First International Workshop* (Washington D.C., USA, 1992), H. B. Anil Nerode, Allen L. Brown Jr. and R. Furuta, Eds. Revised version submitted for publication in the journal Computer and Mathematical Modeling.

[6] BROWN JR., A. L., MANTHA, S., AND WAKAYAMA, T. Exploiting the normative aspect of preference: A deontic logic without actions. *Annals of Mathematics and Artificial Intelligence* (1992).

[7] BROWN JR., A. L., MANTHA, S., AND WAKAYAMA, T. Preference logics and nonmonotonicity in logic programming. In *Logic at Tver, International Conference on Logical Foundations of Computer Science* (Tver, Russia, 1992), A. Nerode, Ed., Springer-Verlag.

[8] BROWN JR., A. L., MANTHA, S., AND WAKAYAMA, T. Preference logics: Towards a unified approach to nonmonotonicity in deductive reasoning. In *Second International Symposium on Artificial Intelligence and Mathematics* (Ft. Lauderdale, Florida, 1992). Revised version to appear in the Annals of Mathematics and Artificial Intelligence.

[9] BROWN JR., A. L., MANTHA, S., AND WAKAYAMA, T. A logical reconstruction of constraint relaxation hierarchies in logic programming. In *ISMIS 93: International Symposium on Methodologies for Intelligent Systems* (1993), Springer-Verlag.

[10] CHELLAS, B. F. *Modal Logic, An Introduction.* Cambridge University Press, Cambridge England, 1980.

[11] FISHBURN, P. Intransitive indifference in preference theory: A survey. *Operations Research 18* (1970).

[12] GOLDFARB, C. F. *The SGML Handbook.* Oxford University Press, Oxford England, 1990.

[13] HALLDEN, S. The logic of better, Lund, 1957.

[14] HANSSON, B. Fundamental axioms for preference relations. *Synthese 18* (1968).

[15] HANSSON, S. O. A new semantical approach to the logic of preference. *Erkenntnis 31* (1989), 1–42.

[16] HUGHES, R. I. G. Rationality and intransitive preferences. *Analysis 40.3* (1980).

[17] HUMBERSTONE, I. L. Inaccessible worlds. *Notre Dame Journal of Formal Logic 24*, 3 (1983), 346–352.

[18] JEFFREY, R. C. *The Logic of Decision.* University of Chicago Press, Chicago, 1983.

[19] LIFSCHITZ, V. Pointwise circumscription: Preliminary report. In *AAAI86* (1986).

[20] MAKINSON, D. General theory of cumulative inference. In *Nonmonotonic Reasoning, Second International Workshop* (New York, 1988), Springer-Verlag, pp. 1–18. Lecture Notes in Artificial Intelligence.

[21] MANTHA, S. First-order preference theories and their applications. Tech. rep., Dept. of Computer Science, University of Utah, 1992.

[22] REITER, R. A logic for default reasoning. In *Readings in Nonmonotonic Reasoning*, M. Ginsburg, Ed. Morgan Kaufmann Publishers, Los Altos California 94022, 1987.

[23] SHOHAM, Y. *Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence*. MIT Press, Cambridge Massachusetts, 1987.

[24] VON WRIGHT, G. H. *The Logic of Preference*. University of Edinburgh Press, Edinburgh Scotland, 1963.

[25] VON WRIGHT, G. H. The logic of preference reconsidered. *Theory and Decision* (1972), 55–67.

[26] WILSON, M., AND BORNING, A. Extending hierarchical constraint logic programming: Nonmonotonicity and inter-hierarchy comparison. In *NACLP* (Cleveland, Ohio, 1989).

Using a Visual Constraint Language for Data Display Specification*

Isabel F. Cruz
Department of Computer Science
Brown University
Providence, RI 02912-1910
ifc@cs.brown.edu

Abstract

In this paper we introduce the *U-term language*, a constraint-based language that has a visual syntax, and allows for the declarative specification of the display of data. Other features of the U-term language include: (1) simplicity and genericity of the basic constructs; (2) ability to specify a variety of displays (pie charts, bar charts, etc.); (3) compatibility with the object-oriented framework of the database language DOODLE.

1 Introduction

In this paper we present a new constraint-based language, the *U-term language*. This language provides:

- A declarative and visual specification of the display of graphical objects with simple and generic constructs.
- The ability to specify a variety of displays such as pie charts, bar charts, and graphs, using Cartesian or polar coordinates.
- Easy integration in an object-oriented framework.

The U-term language is a key component of DOODLE (*Draw an Object-Oriented Database LanguagE*) [Cru92, Cru93a, Cru93b]. The main principle behind DOODLE is that it is possible to ***display and query the database with user-defined pictures***.

A DOODLE visual program is a set of visual rules, which can be read in any order. Visual rules are vertically divided by a double bar. We call the entities to the left and to the right of this bar *D-terms* (for DOODLE terms). The DOODLE program of Figure 1(i) relates to a software engineering application: the graph visualization of the components of a program. In Figure 1(i) there are two kinds of D-terms:

F-terms. These are terms from *F-logic* [KLW90]. F-terms are depicted as strings of characters.

U-terms. These are *user-defined* terms. A U-term is a picture, which is a sentence of the U-term language.

In this program all the terms to the right of the double bar are F-terms, hence the box `f-language`. On the left-hand side, the U-terms are used to define the visual language **softGraph** (for software graph), as indicated by `softGraph`. The program specifies that for the database facts that make the F-terms true, graphical objects similar to the U-terms on the left-hand side will be drawn on the screen. The U-terms are formed of two kinds of symbols: (1) *prototypical symbols*, which specify "by example" the visual attributes (e.g., shape) of the graphical objects to be displayed on the screen, and the spatial relationships between these objects. (In the current example, prototypical symbols happen to be depicted using solid lines.) (2) ***key symbols*** (see Figure 1(ii)) determine how the prototypical symbols are interpreted.

*Partial support was given by ARPA order 8225, ONR grant N00014-91-J-4052.

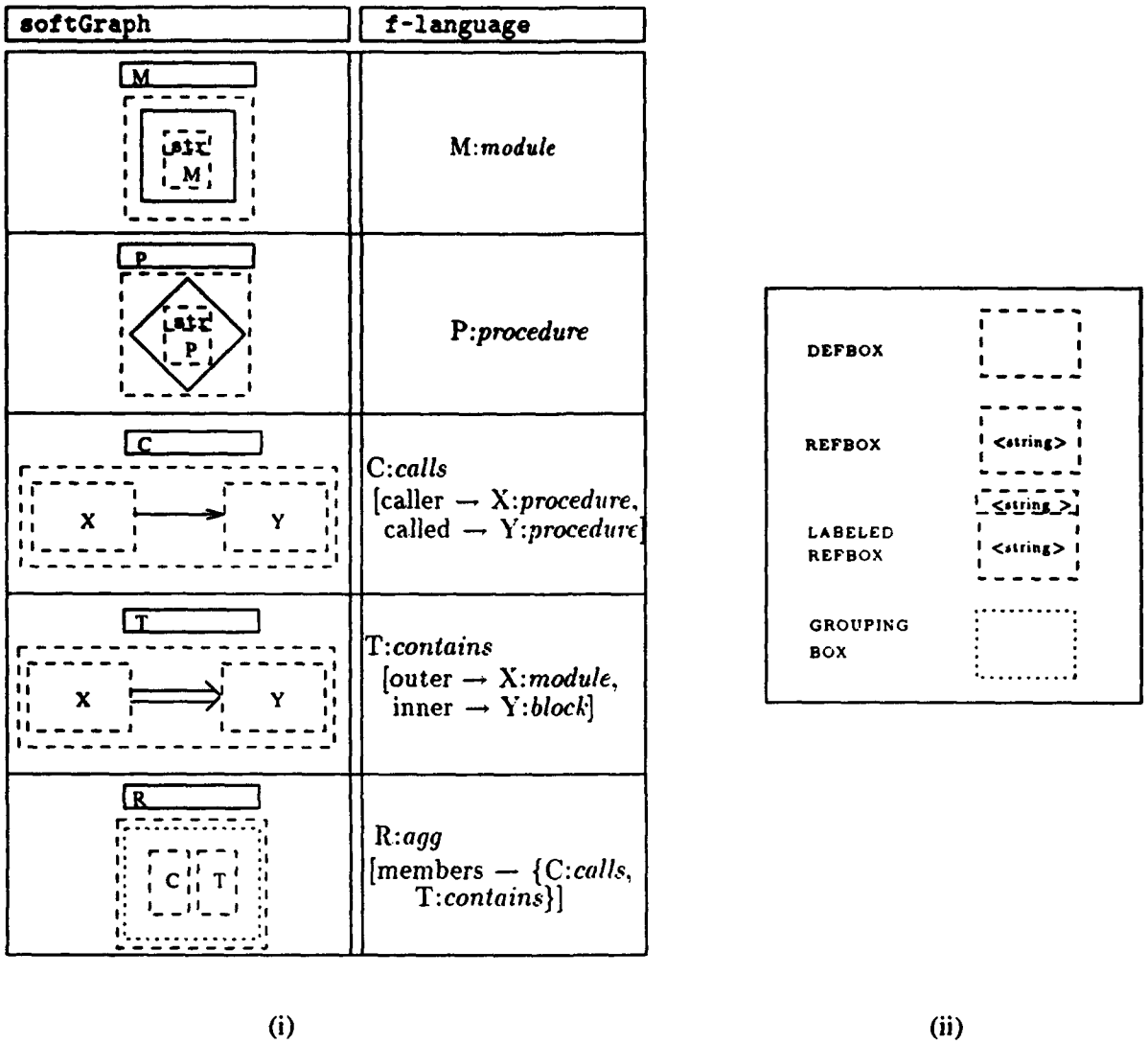


Figure 1: (i) Visual program that defines **softGraph**. (ii) Some key symbols in DOODLE.

The first rule states that any object M in class *module* is to be displayed by a box. The second rule states that any object P in class *procedure* is to be displayed by a diamond. In a similar way, objects C of class *calls* are to be displayed by (simple) arrows, while objects T of class *contains* are to be displayed by double arrows.

The U-term in the third rule specifies the following display: *"Draw a solid arrow that starts on the graphical object that displays database object* X *and ends on the graphical object that displays database object* Y." Therefore, refboxes allow for patterns to refer to patterns that were defined elsewhere. For example, in the objects of class *calls*, the refboxes make reference to the pattern that specifies the display of objects of class *procedure*. This reference is made possible by the use of a defbox in the rule that defines procedure. The defbox indicates the pattern (or part of the pattern) that can be referred to by another rule. In this example the defbox that is defined for objects of class *procedure* includes the prototypical symbol diamond and a labeled refbox. The roles of defbox and refbox are analogous to the concepts of procedure definition and of procedure call. The defbox specifies the display, and the refbox "calls" it.

On the fourth visual rule, the refbox that contains Y can either refer to the defbox in the rule that specifies the display of procedures or to the defbox that specifies the display of modules (note that there is no specification for the display of objects of class *block*, and that *procedure* and *module* are subclasses of *block*).

Labeled refboxes are a generalization of refboxes. While (simple) refboxes make a call to a defbox in the current visual language (**softGraph** in the example), the labeled refboxes make a call to the visual language that labels the refbox. In the first two rules of Figure 1(i) this visualization is **str** (for "string"). This means therefore that M and P are to be displayed using a visualization **str** that is defined by another visual program. The last rule of the visual program states that the visual rendition of a set of objects R is the set of the visual representations of the members of the set (in this case C and T). We use a grouping box to denote the set of visual representations.

The program of Figure 1(i) defines a mapping from database objects to graphical objects on the screen. We call such mapping a *visualization*. The semantics of DOODLE is given by an F-logic program: a visual rule maps to an F-logic rule and U-terms map to F-logic objects. For example, if we assume that the rules of the DOODLE program of Figure 1(i) are the only ones that define the visual language **softGraph**, then we know that (for any database) a graph where some of the nodes are circles is not a picture in **softGraph**. Formally, this stems from the fact that there is no corresponding set of F-logic objects that is a minimal model for the corresponding F-logic program [Cru93a].

This paper is organized as follows. In Section 2 we introduce the graphic principles and the objects and constraints that are the basis for the U-term language. In Section 3 we present the abstract and concrete syntax of the U-term language. Finally, in Section 4 we make a comparison with related work, and discuss topics for future research.

2 Objects and Constraints

We describe graphical objects and the spatial relationships between them using an object-oriented model. This model is the (textual) basis to the U-term language.

2.1 Visual Objects

Visual objects specify graphical objects. They correspond directly to the kinds of graphical objects that the layout program can display and are instances of *visual classes* such as *box*, *circle*, *arrow*, and *text*.

2.2 Landmarks and Anchor Points

Landmarks are used to give dimensions to the objects, and to place objects relatively to other objects. Landmark objects belong to class *landmark*. There are three subclasses of this class: *cartesianLandmark*, *polarLandmark*, and *lineLandmark*. *Anchor point* objects are user-defined landmarks. They belong to the class *anchorpoint*.

Cartesian landmarks. The objects of class *cartesianLandmark* are "mw" (for midwest), "mn" (for midnorth), "me" (for mideast), "ms" (for midsouth), and "c" (for center). We show their position, as located on a box and on a circle in Figure 2.

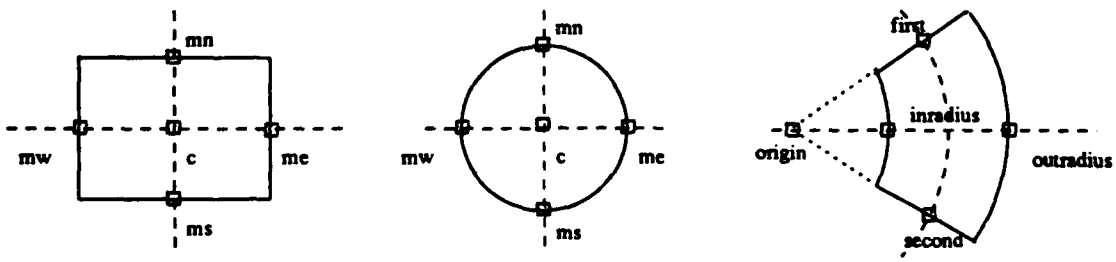


Figure 2: Cartesian landmarks: (i) on a box; (ii) on a circle. Polar landmarks: (iii) on a sector.

Polar landmarks. The objects of class *polarLandmark* are "first" (for first angle), "second" (for second angle), "inradius" (for inner radius), and "outradius" (for outer radius), as shown in Figure 2(iii).

Line landmarks. The objects of class *lineLandmark* are "h" (for head), "t" (for tail), denoting the endpoints of a line.

2.3 Visual Constraint Objects

Visual constraint objects express spatial relationships between one or more visual objects, and are instances of *visual constraint classes*. The constraints are expressed in terms of the objects' landmarks or anchor points. A visual constraint on a single object allows, for example, for one of the dimensions of the object to be specified. A visual constraint between two objects expresses a spatial relationship between two objects. We define two classes: *lengthConstraint* and *overlapConstraint* and describe their types with signature F-logic terms [KLW90].

Length constraint. The F-logic signature of the class *lengthConstraint* is as follows:

$$\begin{aligned} \textit{lengthConstraint}[&\text{firstObj} \Rightarrow \textit{visualObject};\\ &\text{secondObj} \Rightarrow \textit{visualObject};\\ &\text{firstLand} \Rightarrow \{\textit{landmark}, \textit{anchorpoint}\};\\ &\text{secondLand} \Rightarrow \{\textit{landmark}, \textit{anchorpoint}\};\\ &\text{distance} \Rightarrow \textit{realNumberExp};\\ &\text{kind} \Rightarrow \textit{kindType}] \end{aligned}$$

The class *lengthConstraint* has attributes firstObject, secondObject, firstLand, secondLand, distance, and kind. Braces indicate class union. Objects of class *realNumberExp* are either constants, variables, or different kinds of expressions (e.g., $max(X, Y)$, $Height_1 + Height_2$, ≥ 0) of type *real*. Values of attribute *kind* include vertical, horizontal, absolute (Euclidean distance), radial and angular.

The class *lengthConstraint* is: (1) general, since it considers a variety of distances and kinds of distances, and (2) generic, because constraints apply to any objects as long as the landmarks are defined for those objects.

Overlap constraint. Given two visual objects, overlap constraints specify which object is to be drawn on top of the other object. The class *overlapConstraint* has the following signature:

$$\begin{aligned} \textit{overlapConstraint}[&\text{firstObj} \Rightarrow \textit{visualObject};\\ &\text{secondObj} \Rightarrow \textit{visualObject};\\ &\text{top} \Rightarrow\!\!> \textit{visualObject}] \end{aligned}$$

Top indicates which of the two objects is to be displayed on top. The default value for the attribute top is the union of the values for the firstObj and for the secondObj attributes (top is a multi-valued attribute as indicated by $\Rightarrow\!\!>$). This default value works well when the two objects are transparent (e.g., boundaries of rectangles), otherwise, the overlap constraint is not defined. When two objects are specified to overlap, there are two special anchor points named "overlap", one on each object, which will coincide in the resulting picture.

2.4 Examples

Position between two Boxes

Figure 3 shows eight distances that may be defined between the landmarks of two boxes. (Notice that this is a subset of all the possible distances that could be defined by these landmarks). We label these distances with "nn", "ns", "sn", "ss","ww", "we", "ew", "ee". In order to depict and establish completely the spatial constraints between two objects not all the above distances are needed. The examples of Figure 4 illustrate this point. We attach to each picture the distances that are needed to specify the constraints. There will be three objects of class *lengthConstraint* to specify (i) and (iv) and four objects of class *lengthConstraint* to specify (ii) and (iii). In these examples we consider the first graphical object to be the one with smallest identity.

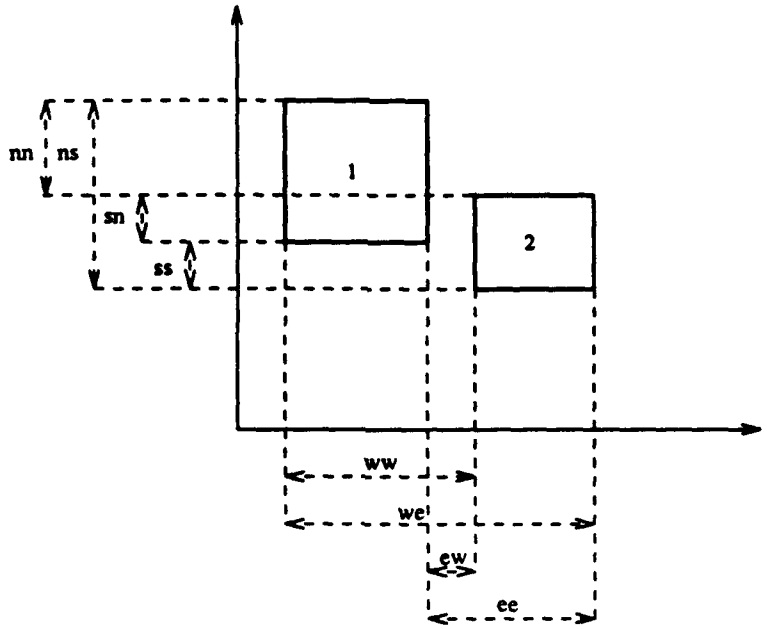


Figure 3: Constraining the position of two objects.

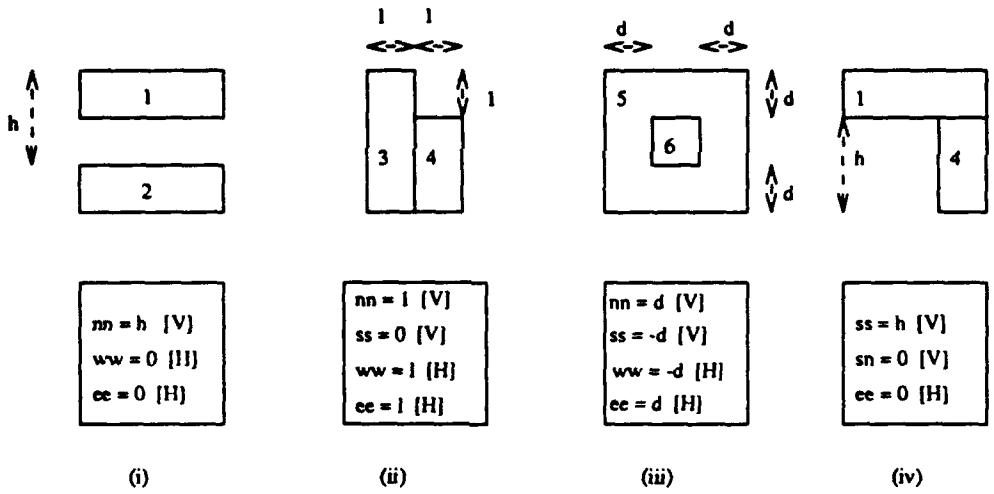


Figure 4: Spatial relationships between pairs of objects ("V" denotes vertical and "H" denotes horizontal).

Position between two Sectors

Figure 5 shows four angular distances and four radial distances that constrain the position of two sectors (these distances are "ff", "fs", "sf", "ss", and "oo", "oi", "io", "ii", where "f" denotes first, "s" denotes second, "i" denotes inner radius and "o" denotes outer radius).

Position of Anchor Points

The following example shows how the position of the anchor point labeled "headpoint" can be specified using an object o_1 of class *lengthConstraint*. In the example the vertical position of the anchor point depends on a landmark in the same object (the object with identity 1).

o_1 : *lengthConstraint*[firstObj → 1;
secondObj → 1;
firstLand → me;

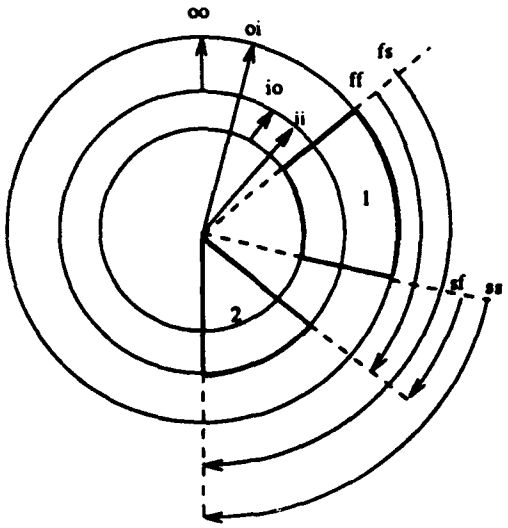


Figure 5: Spatial relationship between pairs of sectors.

secondLand → "headpoint";
distance → d;
kind → vertical]

Positions on the Screen

Vertical and horizontal positions on the screen can be defined by considering the origin to be a visual object.

o_2 : *lengthConstraint*[firstObj → origin;
secondObj → 1;
firstLand → origin;
secondLand → overlap;
distance → d;
kind → vertical]

Overlapping of Objects

The following object describes the overlap of two objects with identities 1 and 2, where object 1 is to be placed on top of object 2. As a result, the two "overlap" anchor points will be made to coincide.

o_3 : *lengthConstraint*[firstObj → 1;
secondObj → 2;
top → 1]

The position of the anchor point "overlap" of the two objects is user-defined. For example, for object 1, it could have been defined as follows:

o_4 : *lengthConstraint*[firstObj → 1;
secondObj → 1;
firstLand → overlap;
secondLand → me;
distance → 0;
kind → absolute]

The absolute position of the overlap point can be specified (see length constraint object with identity o_2).

3 Syntax of the U-term Language

3.1 Abstract Syntax

In the U-term language there are four kinds of symbols: prototypical symbols, key symbols, macro symbols and generic symbols.

Prototypical Symbols

A prototypical symbol consists of:

symbol name. The symbol name uniquely identifies the symbol.

symbol class. Symbol classes include shapes (like box), lines (like straight line), and text.

attribute pairs (attribute name, attribute value). Classes have attributes, whose values specify further the objects. For example the boundary of a box can be solid or dashed. boundary is an attribute name and its attribute values include solid and dashed.

set of landmark pairs (name of landmark, landmark type). Each symbol has a set of landmarks. Landmarks can be of two types: cartesian and polar. Some symbol classes can have landmarks of the two types. Such is the case of circle. There is a special landmark called any, which refers to any landmark on the boundary of the symbol. Its type can be cartesian or polar.

set of anchor point pairs (name of anchor point, anchor point type). This set may be empty. The names of the anchor points are strings chosen by the user. The type of the anchor points can be cartesian or polar.

In Figure 6 we give some examples of prototypical symbols using the syntax of F-logic.

```
b : box[boundary → solid, density → opaque, color → black, texture → plain,
        landmarks → {[name → nw, type → cartesian], ..., [name → C, type → cartesian]},
        anchorpoints → {[name → "headpoint", type → cartesian]}]
s : sector[boundary → dashed, density → transparent, color → black, texture → plain,
          landmarks → {[name → "first", type → polar], ..., [name → "origin", type → polar]},
          anchorpoints → {}]
t : text[value → "Draw", font → roman, size → 12pt]
```

Figure 6: Examples of abstract U-terms for prototypical symbols.

Key Symbols

Table 1 summarizes the syntax of the key symbols in the U-term language. In the table, < ***any symbol but defbox*** > comprises any prototypical symbol, but also any macro symbol or generic symbol that we describe below.

Figure 7 gives examples of abstract U-terms for key symbols. The following observations complement the summarized information in Table 1. A more complete description is in [Cru93a].

- The objects that are values for the attribute contains have to be physically contained in the boxes that form a defbox or a grouping box. This means that if the object overlaps but is not within the boundaries then it is not part of any of those constructs. Note that a defbox cannot contain another defbox, neither can a grouping box.
- origin denotes the point of (0,0) coordinates in the active display area (this is the area where the graphical objects that are specified by the U-term will be displayed).

Symbol Class	Attributes Names	Attribute Value Types
defbox	contains	{< *any symbol but defbox* >}
refbox	name	< *string* > | <*string* > •
labeled refbox	name	< *string* > | < *string* > •
	label name	< *string*>
grouping box	contains	{< *any symbol but defbox*> }
origin	landmark	origin
topright	landmark	topright
length constraint	first object	< *prototypical symbol* > | < *refbox* >
	second object	< *prototypical symbol* > | < *refbox* >
	first landmark	< *landmark* > | < *anchorpoint* >
	second landmark	< *landmark* > | < *anchorpoint* >
	distance	+ (< *expression* >) | − (< *expression* >) | abs (< *expression* >)
	kind	horizontal | vertical | absolute | radial | angular
overlap constraint	first object	< *prototypical symbol* > | < *refbox* >
	second object	< *prototypical symbol* > | < *refbox* >
	top	{ < *prototypical symbol* > | < *refbox* > }

Table 1: Key symbols.

```
d : defbox[contains — b]
l : labeledrefbox[name — "X", labelname — "barChart"]
t : lengthconstraint[firstobject — l, secondobject — l,
                     firstlandmark — ms, secondlandmark — mn,
                     distance — α Y, kind — vertical]
```

Figure 7: Examples of abstract U-terms for key symbols.

- **topright** denotes the point that has greatest X-coordinate (right) and greatest Y-coordinate (top) in the active display area. origin and topright along with the length constraints allow for objects to be placed anywhere in the active display area.

- **distance** can be positive or negative or the absolute value of an expression (the distance is restricted to be absolute when the kind of the length constraint is absolute).

- The kind angular applies only to a pair of polar landmarks or anchor points.

- The kind radial applies to a pair of landmarks or anchor points such that at least one of them is polar. The fact that one of them can be cartesian gives much flexibility to the positioning of polar objects (e.g., sectors). For example, their polar origin can be placed in the cartesian plane. Also text (which has cartesian coordinates) can be placed in a pie chart when related to a polar coordinate.

- The proportionality expression α, allows, for example, to specify that that the height of a bar in a bar chart is proportional to Y, where Y can denote for instance the value of the database attribute "Salary". The syntax of the proportionality expression is as follows:

  < *proportionalityExpression* >:: α <*simpleExpression* > [min :< *num* >][max :< *num* >]
  [sum :< *num* >]

  The value for "max" ("min") is the greatest (smallest) value in the active domain of the specified attribute. "max" and "min" make it easier for the layout program to figure out how to display a range of values within a given screen space. The value of "sum" is the sum of all the values of an attribute. "sum" can be used, for instance, to display pie charts, where it is necessary to know the sum of all the values being represented, so that the sum of the angular widths of each sector (the angular width of each one being proportional to each value) will be made equal to 2π. need not be defined).

Macro Symbols

There are spatial relationships between objects that result from combining two or more length constraint objects. They are not therefore indispensable, but their existence can simplify the user's task of assembling a picture. Next, we present in some detail the macro symbols: contains and grid alignment. Other macro symbols include: Cartesian position (to specify the placement of a graphical object on the plane), and zero distance (to specify that the absolute distance between two landmarks is zero).

Contains

The contains relationship is a macro for four length constraint objects, as outlined in Figure 8. Figure 9,

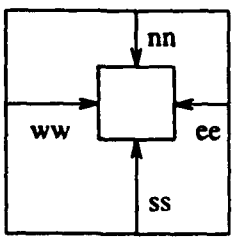


Figure 8: Contains relationship.

where we also give the four length constraint objects it is equivalent to.

C : contains[firstobject — O_1, secondObject — O_2,
nn —$\geq$ 0, ss —$\leq$ 0, ee —$\geq$ 0, ww —$\leq$ 0]
$\equiv$
{Z_1 : lengthconstraint[firstobject — O_1, secondobject — O_2,
firstlandmark — mn, secondlandmark — mn,
distance —$\geq$ 0, kind — vertical],
Z_2 : lengthconstraint[firstobject — O_2, secondobject — O_1,
firstlandmark — ms, secondlandmark — ms,
distance —$\geq$ 0, kind — vertical],
Z_3 : lengthconstraint[firstobject — O_1, secondobject — O_2,
firstlandmark — me, secondlandmark — me,
distance —$\geq$ 0, kind — horizontal],
Z_4 : lengthconstraint[firstobject — O_2, secondobject — O_1,
firstlandmark — mw, secondlandmark — mw,
distance —$\geq$ 0, kind — horizontal]}

Figure 9: Example for contains and equivalent length constraint terms.

Grid alignment

The principle behind grid alignment is the following: if two landmarks or two anchor points or a landmark and one anchor point are on the same horizontal (vertical) line then they have the same horizontal (vertical) coordinates. Figure 10 shows a specification of a bar chart and of two nodes of a linked list. The syntax of this macro symbol is GRID ON. When the alignment is not enforced it corresponds to GRID OFF. In Figure 10 we have emphasized the landmarks for which the vertical or horizontal positions will be enforced.

Generic symbols

Generic symbols are predefined symbols to the layout program. For example, a set of orthogonal axes (as in Figure 11(i)). The parameters for these axes, are the names of two domains, the maximum and minimum values in each of the domains, the space along each axis between each consecutive pair of "labeled ticks" The

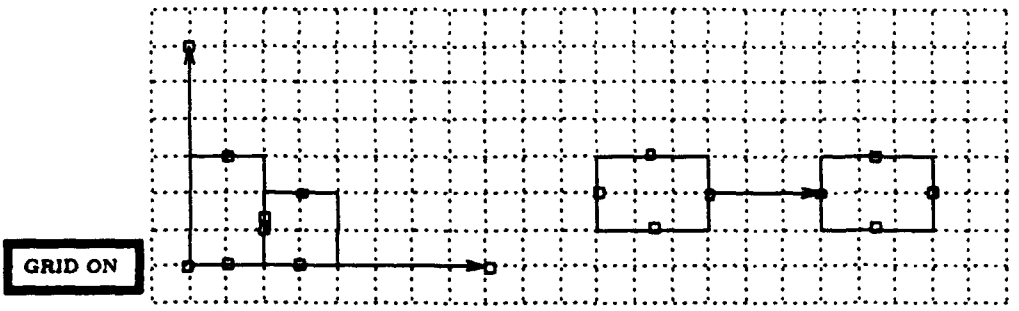


Figure 10: The grid alignment option.

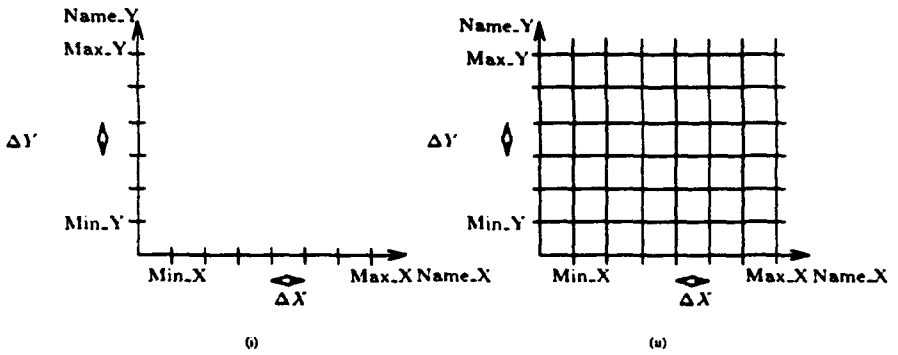


Figure 11: (i) X-Y Axes; (ii) X-Y Grid.

grid generic symbol is similar to axes, but there the "ticks" are replaced by sets of horizontal and vertical lines (see Figure 11(ii)).

3.2 Concrete Syntax

Figure 12 exemplifies (part of) the concrete syntax of the U-term language. In the example, the two solid boxes intersect the grouping box. This is not a syntactic error. The boxes are *not inside* the grouping box, so the latter is actually empty. It is also valid to have a circle inside a refbox. Because there are no syntactic errors, this picture is a well-formed U-term.

The syntactic description of the picture is given in Figure 13. It takes into account the exact position of the graphical objects, as indicated in Figure 12. The abstract syntax is given in Figure 14.

There may be more than one well-formed U-term that is mapped to the same abstract U-term. We can therefore partition the unconstrained U-terms into equivalence classes. We say that two U-terms are *specification equivalent* iff they are described by the same (up to macro symbol equivalence) abstract syntax. Note that this equivalence is stricter than needed, since there may be U-terms that specify the same display, but are not specification equivalent.

4 Conclusions

We have presented a new constraint-based visual language, the U-term language, to specify the display of data. This language is declarative and visual.

Other languages which are also constraint-based, such as IDEAL [Wyk82], [Kam89], and [Gol91], are textual. Of these, only IDEAL addresses the display of sectors like the ones that are used in pie charts (called *wedges*). In ways, our work is closer to (and also drew from) ThingLab [Bor81], in that the user can define visual classes by example, and the language is constraint-based and object-oriented. In ThingLab, constraints are associated with classes. A constraint in a class includes a predicate, which needs to be true for the objects of that class, and methods, which provide alternate ways of satisfying the constraint. In this

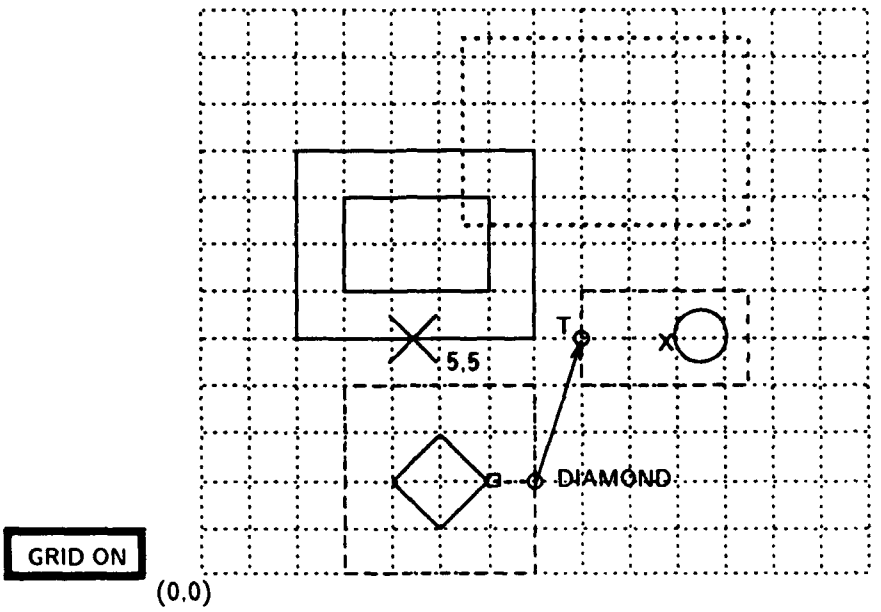


Figure 12: U-term.

box[lx → 2, by → 5, rx → 7, ty → 9, boundary → solid]
box[lx → 3, by → 6, rx → 6, ty → 8, boundary → solid]
groupingbox[lx → 5.5, by → 7.5, rx → 11.5, ty → 11.5]
refbox[lx → 8, by → 4, rx → 11.5, ty → 6, name → "X"]
circle[cx → 10.5, cy → 5, r → 0.5, boundary → solid]
defbox[lx → 3, by → 0, rx → 7, ty → 4]
diamond[lx → 4, by → 1, rx → 6, ty → 3, boundary → solid]
cartesianposition[cx → 4, cy → 5, label → "5,5"]
anchorpoint[cx → 8, cy → 5, name → "T"]
anchorpoint[cx → 7, cy → 2, name → "DIAMOND"]
zerolength[lx → 6, by → 2, rx → 2, ty → 7]
arrow[lx → 7, by → 2, rx → 8, ty → 5, boundary → solid]
gridon

Figure 13: Description of a U-term.

paper we are concerned with the predicate (visual) specification, but we are not addressing the satisfaction of constraints. Future work will focus on the following topics:

Language design. Evaluation of the U-term language in the specification of a variety of 2D displays. In addition, we would like to test the robustness of the key symbols with different sets of primitive symbols (e.g., for 3D display).

Constraint query languages. Constraint query languages such as [KKR90] have a textual syntax. The U-term language could provide a visual syntax for such languages.

Graph drawing. Identification of the kinds of graph layouts [DET93] that can be specified by the U-term language and by DOODLE; characterization of the graph properties (e.g., planarity) that can be expressed.

Design of the interface. Figures 1(i) and 12 show different levels of detail in the presentation of the U-terms. The design of the interface so that the user can choose different presentations of the U-terms is an interesting topic.

```
a : box[boundary → solid]
b : box[boundary → solid]
c : groupingbox[contains → {}]
d : refbox[name → "X"]
e : circle[boundary → solid]
f : defbox[contains → {g}]
g : diamond[boundary → solid]
l : arrow[boundary → solid]
n : cartesianposition[firstObject → a, landmark → ms, x_position → 5, y_position → 5]
o : lengthconstraint[firstObject → g, secondObject → l, firstLandmark → me, secondLandmark → t,
                     distance → 0, kind → absolute]
p : lengthconstraint[firstObject → d, secondObject → l, landmark → "T", landmark → h,
                     distance → 0, kind → absolute]
r : lengthconstraint[firstObject → b, secondObject → g, firstLandmark → me,
                     secondLandmark → me, distance → 0, kind → horizontal]
s : lengthconstraint[firstObject → a, secondObject → e, firstLandmark → ms,
                     secondLandmark → "T", distance → 0, kind → vertical]
u : contains[firstObject → a, secondObject → b,
             nn →≥ 0, ss →≤ 0, ww →≥ 0, ee →≤ 0]
v : contains[firstObject → d, secondObject → e,
             nn →≥ 0, ss →≤ 0, ww →≥ 0, ee →≤ 0]
```

Figure 14: Abstract U-term.

Expressive power of DOODLE as a picture generator. The example of Figure 1(i) is relatively simple: for instance we have no recursion in the rules. With recursion it seems that DOODLE would have at least the same expressive power to generate pictures as visual multiset grammars [Gol91]. The precise comparison of the expressive power of both approaches is another subject for future research.

Acknowledgements Thanks to Theo Norvell for fruitful discussions that led to the definition of the visual constraint objects and to the refinement of the abstract syntax.

References

[Bor81] Alan Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[Cru92] Isabel F. Cruz. DOODLE: A Visual Language for Object-Oriented Databases. In *ACM-SIGMOD Intl. Conf. on Management of Data*, pages 71–80, 1992.

[Cru93a] Isabel F. Cruz. *Querying Object-Oriented Databases with User-Defined Visualizations*. PhD thesis, Department of Computer Science, University of Toronto. 1993. To appear.

[Cru93b] Isabel F. Cruz. User-defined Visual Languages for Querying Data. Manuscript, April 1993. Dept. of Computer Science, Brown University.

[DET93] Giuseppe Di Battista, Peter Eades, and Roberto Tamassia. Algorithms for Drawing Graphs: an Annotated Bibliography. Technical report, Department of Computer Science, Brown University, March 1993.

[Gol91] Eric J. Golin. A Method for the Specification and Parsing of Visual Languages. Technical Report CS-90-19, Brown University, May 1991.

[Kam89] Tomihisa Kamada. *Visualizing Abstract Objects and Relations - A Constraint-Based Approach*. World Scientific, Singapore, 1989.

[KKR90] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Contraint Query Languages. In *ACM Symposium on Principles of Database Systems*, pages 299–313, 1990.

[KLW90] Michael Kifer, Georg Lausen, and James Wu. Logic Foundations of Object-Oriented and Frame-Based Languages. Technical Report 90/14 (2-nd revision), Department of Computer Science, SUNY Stony Brook, 1990.

[Wyk82] Christopher J. Van Wyk. A High-Level Language for Specifying Pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.

Constraint Management in a Declarative Design Method for 3D Scene Sketch Modeling

Stéphane Donikian
IRISA
Campus de Beaulieu
35042 Rennes Cedex, FRANCE
donikian@irisa.fr

Gérard Hégron
Ecole des Mines de Nantes
3 rue Marcel Sembat
44049 Nantes cedex 04, FRANCE
hegron@emn.fr

Abstract

In this paper, we present a dynamic model associated with an intelligent CAD system aiming at the modeling of an architectural scene sketch. Our design methodology has been developed to simulate the process of a user who tries to give a description of a scene from a set of mental images. The scene creation is based on a script which describes the environment from the point of view of an observer who moves across the scene. The system is based on a declarative method viewed as a stepwise refinement process. For the scene representation, a qualitative model is used to describe the objects in terms of attributes functions, methods and components. The links between objects and their components are expressed by a hierarchical structure, and a description of spatial configurations is given by using locative relations. The set of solutions consistent with the description is usually infinite. So, either one scene consistent with this description is calculated and visualized, or reasons of inconsistency are notified to the user. The resolution process consists of two steps: firstly a logical inference checks the consistency of the topological description, and secondly an optimization algorithm deals with the global description and provides a solution. Two examples illustrate our design methodology and the calculation of a scene model.

1 Introduction

In the current literature, several approaches have been attempted to design CAD systems. A first approach consists of an extension of classical CAD systems with the help of parametric objects or variational geometry [1], but models required by those systems are still very close to the traditional geometrical models and imply a bottom up design methodology by using imperative methods. Furthermore, this approach is not suitable when the design scene is complex or when the designer thinks about his project in a more semantical way. A second approach consists in building expert systems; these systems tend to focus on a particular domain and are only useful for routine design in a well known application domain completely formalized by a set of rules and constraints [2, 3]. Both approaches are far from architect's considerations during the first stage of design which is more a top down approach and a stepwise refinement process. A third approach met a constantly increasing interest in all CAD domains [4, 5, 6, 7, 8, 9]. This approach is more declarative and offers to the designer a more progressive and dynamic scene specification, leaving to the system the care of suggesting solutions, detecting inconsistencies and manipulating incomplete knowledge. An important difference between declarative and imperative methods (figure 1) is that a declarative method does not provide a unique solution like an imperative method does, but provides a model corresponding to a large number of scenes for which the system proposes one or more solutions.

Our system is based on the third approach and offers to the designer the ability to describe the topology and geometry of the scene in a declarative way by means of properties and constraints on objects and their spatial configuration. This system deals with under-constrained and over-constrained scene descriptions and proposes to the designer one of the numerous scenes whenever the description is consistent, or explicits the inconsistency reasons. The goal of this paper is to put an emphasis on the different sorts of properties and constraints our system can manage, how they are performed and how a particular solution is calcuted from the global description. Other parts are not detailed but are referred to previous papers. The next section

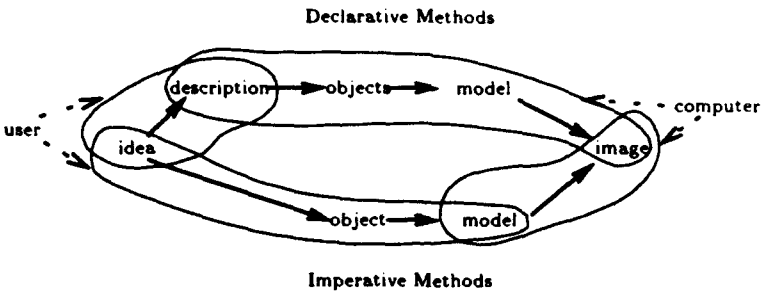


Figure 1: Declarative and imperative methods

gives an overview of our system structure. Section three deals with the two steps involved in the resolution process and section four gives some details about the different kinds of inconsistencies which may occur during the design process. Finally some implementation details and illustrations of our method are given.

2 System Overview

We intend to apply the declarative methodology to architectural design. Our goal is not to get the final and precise geometric model of each component of the scene but to assist the designer in creating sketches meeting the main properties and constraints of mental images. The architectural project design we will try to simulate is based on a script which describes the scene from the point of view of a user who moves across the scene [10]. From each viewpoint, the user increases the scene description by adding new objects and new properties and constraints about objects and their spatial configurations, and a current model of the scene is then proposed by the system. For architectural design, the complexity comes from the spatial configuration of objects and from the large amount of data. This observation explains why we are not interested in defining complex geometrical objects but in providing a high level description of object characteristics and spatial configurations.

Our model is composed of a 3-tuple <**O**, **S**, **L**>, where **O** is the set of scene objects, **S** the scene hierarchical structure and **L** the set of locative relations and constraints between objects. A dynamic and interactive user interface is used to offer a high level interaction to the designer, and the dialogue is carried out with the help of an object oriented language (figure 2). The verification of the scene description consistency and the proposition of a scene (solution of the description) are made during the resolution process.

Each class of objects is a member of an architectural abstraction level (urban, building, architectural space, architectural element, architectural components, architectural constructors). The hierarchical structure of the scene is represented by a directed acyclic graph whose nodes are objects and whose arcs link an object to another one as a *part of* or as a *copy of*. Each object refers to a particular class, and a class is composed of attributes, functions, methods and a prototype. Attributes denote characteristics of the object which can be represented by a variable of atomical or numerical type. A function defines the object characteristics represented by an analytical inequation which is composed by:

- usual arithmetic operators +, -, *, /,
- parenthesis (,),
- real constants,
- basic geometrical constraints like volume, surface area, segment length, ...
- numerical attributes of the object and its components,
- functions of the object and of its components.

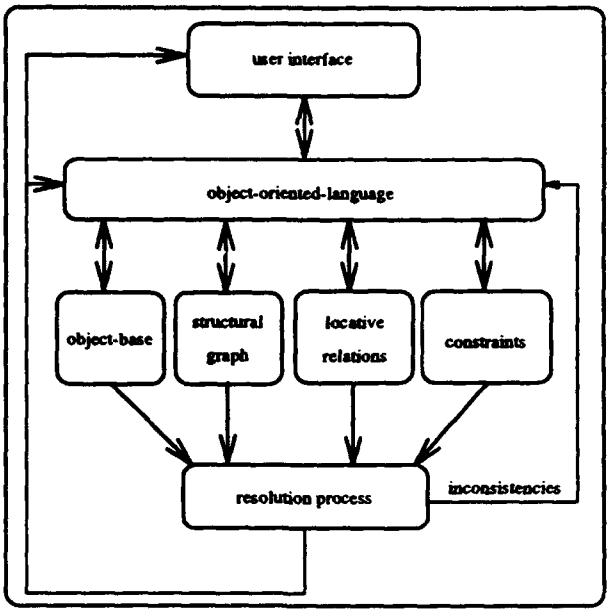


Figure 2: The system architecture

A geometrical constraint will be represented by a function of the object from which all parameters belong to a part of itself. Methods correspond to qualifying adjectives, allow to refine the description of an object, and are described by a sequence of predicates. Some attributes are predefined for every class of objects and represent characteristics useful for locative relation management and for the scene visualization. Each object is defined in its own reference system of axes and is included in its bounding box with edges parallel to the axes. Dimensions of the bounding box are either fixed, or included in a possible value domain, or unknown, and its position depends on locative relations and geometrical constraints.

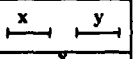
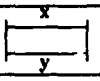
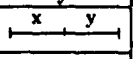
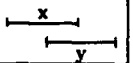
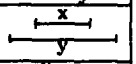
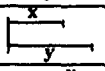
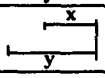
A locative expression involves a locative prepositional phrase together with whatever the phrase modifies [11]. Each locative expression is given in accordance with the current locator's viewpoint. In our system, a locative expression is composed of a locative syntagma, a verb (to be) and two noun phrases: N_{target} is *locative syntagma* N_{site}. The subject (N_{target}) refers to the located entity and the object (N_{site}) to the reference entity. Each locative syntagma possesses its own semantic defined in the relation reference system. The orientation of this reference system depends on the site orientation and for each locative expression, the site can have three kinds of orientation [12] (intrinsic, deictic and contextual orientations). The description is given for a succession of locator's viewpoints, and for each of them the locator participates to the scene description.

Locative relations are given on object bounding boxes, and the locative syntagma semantic [13] is expressed by one constraint along each axis of the relation reference system between edges corresponding to the projection of object bounding boxes (figure 3). Each constraint $C(X, Y)$ corresponds to a disjunction of some Allen's elementary relations [14], which allows to describe the relative positionning of the two segments X and Y along an axis:

$$C(X, Y) = (\vee_{i=1}^{13} \alpha_i . r_i(X, Y)) \text{ with } \alpha_i \in \{0, 1\} \text{ and } r_i \text{ one of the thirteen Allen's relations (figure 3)}$$

Other locative relations are given on object's sides. Each side is represented along the axis parallel to its normal by a projection point of the side on the axis. The relative positionning of two parallel sides is expressed by a constraint on corresponding points (figure 4). A constraint $c(x, y)$ between two points x and y along an axis is expressed by:

$$c(x, y) = (\vee_{i=1}^{3} \alpha_i . r_i(x, y)) \text{ with } \alpha_i \in \{0, 1\} \text{ and } r_i \in \{precede, identical_to, follow\}$$

Relation	Symbol	Symbol for inverse	Illustration
x before y	<	>	
x equal y	=	=	
x meets y	m	mi	
x overlaps y	o	oi	
x during y	d	di	
x starts y	s	si	
x finishes y	f	fi	

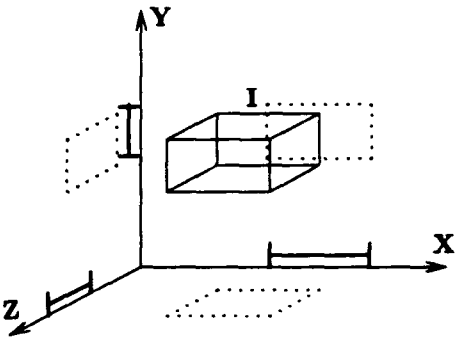


Figure 3: The thirteen relations between two intervals and the bounding box projection along the three axes

Relation	Symbol	Illustration
x precede y	*precede*	
x identical to y	*identical_to*	
x follow y	*follow*	

Figure 4: The three possible relations between two points

To have a unique representation for this two kinds of constraints, each segment X is represented by its two extremities X_b and X_e which correspond to the beginning and end points of the segment. Each Allen's relation $r_i(X, Y)$ between the two segments X and Y is then expressed in the following form:

$$r_i(X, Y) \equiv precede(X_b, X_e) \wedge precede(Y_b, Y_e) \wedge r_{i_1}(X_b, Y_b) \wedge$$

$$r_{i_2}(X_b, Y_e) \wedge r_{i_3}(X_e, Y_b) \wedge r_{i_4}(X_e, Y_e)$$

with $r_{i_j} \in \{precede, identical_to, follow\}$.

A description being a conjonction of constraints, each Allen's constraint must be expressed in the following disjunctive normal form to make the union of all constraints on the same segments or extremities:

$$C(X, Y) = [(\wedge_{j=1}^{4}(\vee_{k=1}^{3}\alpha_{k_j}.r_k(X_{u_j}, Y_{v_j}))) \wedge precede(X_b, X_e) \wedge precede(Y_b, Y_e)]$$

where $\alpha_{k_i} \in \{0, 1\}$, $(u_j, v_j) \in \{(b, b), (b, e), (e, b), (e, e)\}$, $r_k \in \{precede, follow, identical_to\}$

Among all Allen's constraints there are only 187 wich are expressible in this form [15]. Constraints expressible in a disjunctive normal form correspond to constraints which present a continuity in their geometrical configuration area. For example the locative relation *The chair is on the left or on the right of the table* is not expressible in a disjunctive normal form on the lateral axis. On the other

hand, the locative relation *The box X is on the table Y* corresponds to $C_{vertical}(X, Y) = \{m_i\}$ and $C_{frontal}(X, Y) = C_{lateral}(X, Y) = \{o \vee o_i \vee d \vee d_i \vee s \vee s_i \vee f \vee f_i \vee =\}$ is expressible in the following normal form:

- $C_{frontal}(X, Y) = [(X_b \; precede \; X_e) \wedge (Y_b \; precede \; Y_e) \wedge (X_b \; precede \; Y_e) \wedge (Y_b \; precede \; X_e) \wedge (X_b \; \{precede \vee follow \vee identical\ to\} \; Y_b) \wedge (X_e \; \{precede \vee follow \vee identical\ to\} \; Y_e)]$
- $C_{lateral}(X, Y) = [(X_b \; precede \; X_e) \wedge (Y_b \; precede \; Y_e) \wedge (X_b \; precede \; Y_e) \wedge (Y_b \; precede \; X_e) \wedge (X_b \; \{precede \vee follow \vee identical\ to\} \; Y_b) \wedge (X_e \; \{precede \vee follow \vee identical\ to\} \; Y_e)]$
- $C_{vertical}(X, Y) = [(X_b \; precede \; X_e) \wedge (Y_b \; precede \; Y_e) \wedge (X_b \; identical\ to \; Y_e) \wedge (Y_b \; precede \; X_e) \wedge (X_b \; follow \; Y_b) \wedge (X_e \; follow \; Y_e)]$

3 The Calculation of a Scene Description Solution

At any stage of the design process, a solution s, among the infinite set of valid solutions S, can be calculated and visualized. The solution calculation is achieved in two steps: the first step allows to check the consistency of the locative description and to obtain a minimal system of linear inequations; the second step adds to the previous system the set of geometrical constraints (object functions) and proposes one solution of the global description by minimizing an objective function subject to the global system of linear and nonlinear inequalities.

From the set of locative relations, a directed valuated graph is constructed for each axis[16], whose nodes are edge extremities of the bounding box. Graph arcs represent the relative positionning of points by using only $\{precede\}$ and $\{precede \vee identical_to\}$ relations.

$r_i(X_k, Y_l)$	corresponding arc
$\{precede\}$	$X_k \xrightarrow{\mathbf{les}} Y_l$
$\{identical_to\}$	$X_k \xrightarrow{\mathbf{eq}} Y_l$ $Y_l \xrightarrow{\mathbf{eq}} X_k$
$\{follow\}$	$Y_l \xrightarrow{\mathbf{les}} X_k$
$\{precede \vee identical_to\}$	$X_k \xrightarrow{\mathbf{leq}} Y_l$
$\{follow \vee identical_to\}$	$Y_l \xrightarrow{\mathbf{leq}} X_k$
$\{precede \vee follow\}$	no arc creation
$\{precede \vee follow \vee identical_to\}$	no arc creation

Figure 5: Correspondance between constraints and graph arcs

There is no creation of an arc connecting two nodes when the disjunctive relation is $\{precede \vee identical_to \vee follow\}$ or $\{precede \vee follow\}$, because the relative positionning of these two nodes is considered as being free. This description is logically consistent if there is at least one solution to order extremities of segments. For example, the set of constraints $(Y \; \mathbf{d} \; X)$, $(X \; \mathbf{m} \; Z)$, $(Y \; \mathbf{o} \; Z)$ is logically inconsistent, because points X_e, Y_e and Z_b cannot be ordered ($X_e \overset{\mathbf{eq}}{\leftrightarrow} Z_b \overset{\mathbf{les}}{\rightarrow} Y_e \overset{\mathbf{les}}{\rightarrow} X_e$). The consistency of the description is checked (in a unique graph traversal) by searching for a circuit.

Each arc is also valuated by the information of its length, which is given by bounding box dimensions or by distance constraints between the object sides. Each distance between two nodes is either fixed, or included in a possible value domain, or unknown :

- $\xrightarrow{d}$: for a fixed distance **d**,
- $\xrightarrow{[a,b]}$: for a distance in a possible value domain **[a,b]**,

- $\xrightarrow{nil}$: for an unknown distance.

For each arc from X_k to Y_l whose length is unknown, if there is a path between its two nodes of length higher than one, then the arc is removed by applying antitransitivity rule. After this reduction process, some of the arc lengths are still unknown, and it would be interesting to reduce their possible value domain by propagation of known values, but the global propagation algorithms have a Non-deterministic Polynomial complexity. On the other hand, we can apply local propagation algorithms, according to the local graph structure, with a weak cost.

If no inconsistency has been detected at the preceding step, then a minimal system of linear inequations is obtained from the three graphs. Each arc from X_k to Y_l with *les* or *leq* valuation is corresponding to one of the three following expressions:

1. $(X_k \xrightarrow{d} Y_l) \Rightarrow (V(Y_l) - V(X_k)) = d$
2. $(X_k \xrightarrow{[a,b]} Y_l) \Rightarrow a \leq (V(Y_l) - V(X_k)) \leq b$
3. $(X_k \xrightarrow{nil} Y_l) \Rightarrow 0 \leq (V(Y_l) - V(X_k)) \leq L$ (L is the domain length along the corresponding axis)

Each graph node X_k is performed by an expression (by using the function V) corresponding to the sum of a variable and a constant as follows:
$\forall X_k$,

- $\forall Y_l \in pred(X_k)$, $(Y_l \xrightarrow{[a,b] \vee nil} X_k) \Rightarrow V(X_k) = new\,variable\,(x_i)$
- $\exists Y_l \in pred(X_k)$, $(Y_l \xrightarrow{d} X_k) \Rightarrow V(X_k) = V(Y_l) + d$

The function $pred(X)$ defines the set of the direct predecessors of the node X. This allows to reduce the number of variables and inequalities, and also to detect some numerical inconsistencies as for the following system:

$$\left.\begin{array}{l} 30 \leq V(j) - V(k) \leq 50 \\ V(j) = V(i) + 40 \\ V(k) = V(i) + 20 \end{array}\right\} \Longrightarrow 30 \leq 20 \leq 50$$

Our goal is not to detect all inconsistencies of the linear system but only those which can be easily identified during the system construction. Some geometrical constraints are linear, like relations between segment lengths, and are added to the previous linear system; the others form a vector of nonlinear inequations. This system can be well-constrained, but it is generally under or over-constrained: thus it is not possible to enumerate all solutions of the description. The translation of the problem into a system of linear and nonlinear inequalities allows us to solve geometrical and locative constraints simultaneously and to propose one solution among the infinity of possible solutions (when the description is consistent). This system is described by the following equations :

1. $\forall i \in \{1, \ldots, n\}$, $l_i \leq x_i \leq u_i$,
2. $\forall j \in \{n+1, \ldots, p\}$, $l_j \leq \sum_{i=1}^{n} \alpha_i^j . x_i \leq u_j$,
3. $\forall r \in \{p+1, \ldots, q\}$, $l_r \leq C_r(x_1, \ldots, x_n) \leq u_r$.

The system is solved by using a resolution algorithm which minimizes a quadratic objective function F as follow:

$$minimize\,F(x)(x \in R^n)\,subject\,to\,:\, l \leq \left\{\begin{array}{l} x \\ A_L x \\ C(x) \end{array}\right\} \leq u$$

where $x = (x_1, \ldots, x_n)$, A_L is an $(p - n)$ by n constant matrix, and $C(x)$ is an $(q - p)$ element vector of nonlinear constraint functions. A numerical solution of the problem is computed by using a NAG[1] library routine, essentially similar to the subroutine SOL/NPSOL described in Gill et al [17]. The objective function F to be minimized is given by the following equation:

$$F(x_1, \ldots, x_n) = \sum_{i=n+1}^{p} \beta_i . (\frac{(u_i + l_i)}{2} - (x_k - x_l))^2,\ k, l \in \{1, \ldots, n\},$$

where $\beta_i = 1$ if the (i-n) linear equation is in the form $l_i \leq x_k - x_l \leq u_i$, and if $(l_i \neq u_i)$ else $\beta_i = 0$. This function allows to keep the equilibrium of the object's space distribution.

4 Consistency of the Description

At every moment of the design process, an inconsistency can be generated by the description. Firstly we have topological inconsistencies due to an incompatibility between some locative relations. Secondly we may have some geometric inconsistencies: for instance when different values are given for the depth and width of an objet while one says this object is cubic. A part of this second kind of inconsistency can be detected during the construction of the global system and will be also easily expressed to the user. Other inconsistencies are detected during the numeric resolution process.

5 System Implementation and Results

A first version of this system has been carried out[10]. The object-oriented-language and knowledge representation model have been written in Quintus-Prolog and C++. The user interface is composed of a graphic zone written in Phigs included in a Motif window manager environment. The routine of the NAG library, which performs the numerical resolution, is directly called by prolog. We use a global illumination rendering algorithm [18] to obtain realistic images of the scene. The following examples illustrate our method.

A first pedagogic example shows the management of objects whose some dimensions are unknown, and on which the user gives some constraints. The scene is composed of a carpet, a table, two chairs and the following description:

- *The carpet is two centimeters high, five meters wide and three meters deep.*
- *The table is seventy centimeters high and its width and depth are unfixed.*
- *Both chairs are one meter high and their width and depth are unfixed.*
- *The carpet is in front of the locator.*
- *The table and the two chairs are on the carpet.*
- *Both chairs are respectively on the left and on the right of the table.*

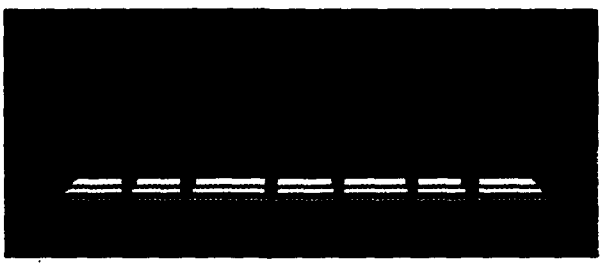


Figure 6: Initial solution of the first example

The scene solution calculated by our system is shown on the figure 6. Adding a constraint defining that *the table depth is twice more important than the width of both chairs* modifies the scene only on the locator



Figure 7: *The table depth is twice more important than width of both chairs*

frontal axis (figure 7) because chairs are intrinsically oriented.

Figure 8 is obtained after adding a constraint defining that *a quarter of the carpet area is occupied by both chairs and the table.*

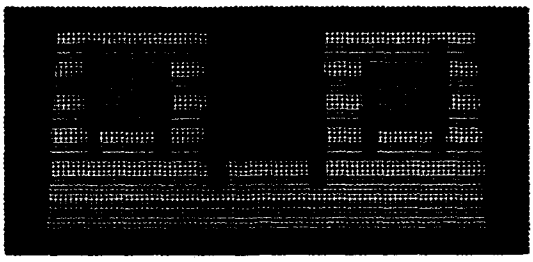


Figure 8: *A quarter of the carpet area is occupied by both chairs and the table*

The scene illustrated by figure 9 has been produced after the definition of the table width (two meters and eighty centimeters). In this scene, the area occupied by the chairs has decreased proportionnaly to the increase of the table area, maintaining the length relation on the locator frontal axis.

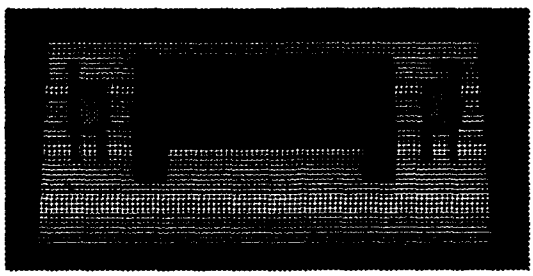


Figure 9: *The table width is equal to two meters and eighty centimeters*

This example has shown that locative relations and geometrical constraints are jointly taken into account in our system, which allows us to constantly offer to the designer a solution of the global description as long as the scene description is consistent.

Through an easy description of a building, the second example illustrates that our system is useful even for more complex scenes. This building (figure 10) is composed of a ground floor, four identical storeys and a roof. On the front side of the building, each storey is composed of five identical rooms: there are two identical apertures in the front wall of rooms, in which a window is embedded. The ground floor is composed of four identical pillars, a winding stair and two rooms.

[1]NAG is a registered trademark of: the Numerical Algorithms Group Limited and The Numerical Algorithms Group Incorporated.

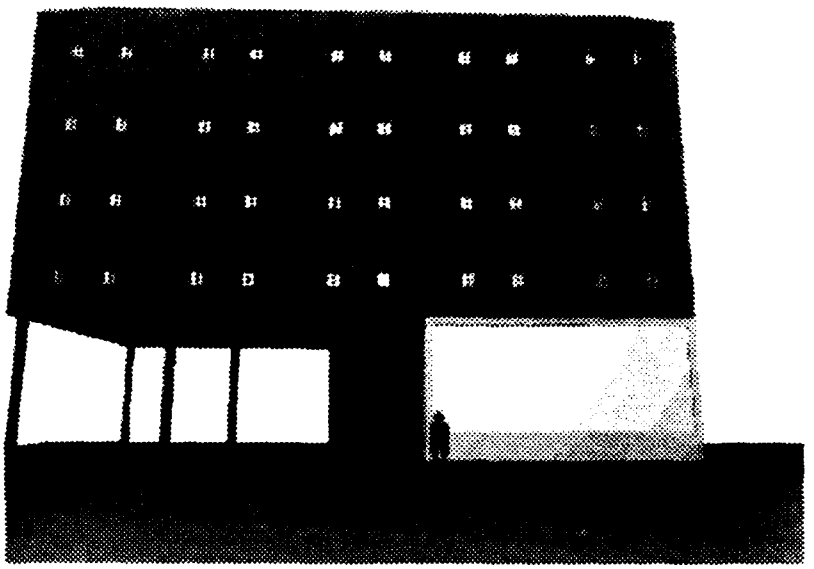


Figure 10: A building example

Conclusion

This paper has presented in detail the management of constraints and properties in our declarative design methodology which provides the ability to manipulate uncertainty for both numerical and locative constraints. The description of objects at a level higher than the geometrical one and the use of prototypes permit to reduce the decomposition/recomposition step. Examples illustrate the capability of our system to jointly deal with a topological and geometrical knowledge. One of its main characteristics is the cooperation between two inference techniques which are most of the time competing: on the one hand a logical inference stemming from Allen's temporal logic and instant logic, on the other hand a numerical inference based on an optimization under constraints algorithm. We are currently extending this model to more complex geometrical objects, and are investigating other design domains involving new kinds of constraints.

References

[1] R. Light and D. Gossard. Modification of geometric models through variational geometry. *Computer Aided Design*, 14(4):209–214, July 1982.

[2] U. Flemming, R.F. Coyne, T. Glavin, H. Hsi, and M. Rychener. *A Generative Expert System for the Design of Building Layouts(Final Report).* Technical Report EDRC 48-15-89, Carnegy Mellon University, 1989.

[3] P. Quintrand, J. Zoller, R. de Filippo, and S. Faure. A model for the representation of urban knowledge. *planning and design*, 18(1):71–83, 1991.

[4] P.J.W. ten Hagen and T. Tomiyama (Eds.). *Intelligent CAD Systems I. Theoretical and methodological aspects.* Springer-Verlag, 1987.

[5] M. Lucas. Equivalence classes in object space modelling. In T. L. Kunii, editor, *Proc. of Working Conference on Modeling in Computer Graphics*, pages 17–34, IFIP TC 5/WG 5.10, Springer Verlag, Tokyo, Japan, April 1991.

[6] J.S. Gero (Eds.). *Artificial Intelligence in Design.* Springer-Verlag, 1989.

[7] E. Lang, K.U. Carstensen, and G. Simmons. *Modelling Spatial Knowledge on a linguistic Basis. Lecture Note in Artificial Intelligence*, Springer-Verlag, 1991.

[8] F. Giunchiglia and E. Trucco. *Object Configuration by Incremental Ill-described Spatial Constraints.* Technical Report DAI Research Paper NO. 400, Department of Artificial Intelligence, University of Edinburgh, 1988.

[9] P. Veerkamp and P.J.W. ten Hagen. Qualitative reasoning about design objects. In *5th International Conference on the Manufacturing Science and Technology of the Future*, Enschede, Pays-Bas, June 1991.

[10] S. Donikian. *Une approche déclarative pour la création de scènes tridimensionnelles : application à la conception architecturale.* PhD thesis, University of Rennes I, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, December 1992.

[11] A. Herskovits. *Language and spatial cognition.* Cambridge University Press, 1986.

[12] M. Aurnague. *Contribution à l'étude de la sémantique formelle de l'espace et du raisonnement spatial : la localisation interne en français, sémantique et structures inférentielles.* PhD thesis, Université Paul Sabatier, IRIT, 118, route de Narbonne, 31062 Toulouse Cedex, February 1991.

[13] S. Donikian and G. Hégron. A declarative design method for 3d scene sketch modeling. To appear in *EUROGRAPHICS'93 Conference Proceedings*, Barcelona, Spain, September 1993.

[14] J.F. Allen. An interval based representation of temporal knowledge. In *Proceedings of the seventh International Joint Conference on Artificial Intelligence*, pages 221–226, August 1981.

[15] T. Granier. *Contribution à l'étude du temps objectif dans le raisonnement.* Technical Report 716-I-73, IMAG, Février 1988.

[16] S. Donikian and G. Hégron. The kernel of a declarative method for 3d scene sketch modeling. In *Graphicon'92*, to appear in Programming and Computer Science, Moscow, Russia, September 1992.

[17] P.E. Gill, S.J. Hammarling, W. Murray, M.A. Saunders, and M.H. Wright. *User's Guide for LSSOL (version 1.0).* Technical Report SOL 86-1, Department of Operations Research, Stanford University, 1986.

[18] K. Bouatouch and P. Tellier. A two-pass physics-based global illumination model. In *Proceedings of Graphics Interface 92*, Vancouver, Canada, May 1992.

The Geometry in Constraint Logic Programs

Thomas Dubé *
dube@cs.nyu.edu

Chee-Keng Yap †
yap@cs.nyu.edu

Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, New York, 10012

Abstract

Many applications of constraint programming languages concern geometric domains. We propose incorporating strong algorithmic techniques from the study of geometric and algebraic algorithms into the implementation of constraint programming languages. Interesting new computational problems in computational geometry and computer algebra arises from such considerations. We look at what is known and what needs to be addressed.

1 Introduction

It is now widely recognized that the programming paradigm of constraint satisfaction is a fundamental one which raises new issues for the programming language community. Such issues are covered under the rubric of "logic programming" and include new language constructs, logical issues, and programming semantics and pragmatics. The book of DeGroot and Lindstrom [8] is a representative of such concerns. In the general setting, the focus is mostly on syntactic properties of substitution and equality[1], with associated algorithmic problems such as unification.

It is no surprise that the deepest applications of constraint satisfaction occurs in very classical domains of mathematics. Some seminal papers in constraint programming (e.g., [1,29,20]) hint at such domains by way of examples. These domains have been investigated by algebraists and geometers for centuries, and include various subrings of the complex field $\mathbf{C}$ and the affine or projective spaces over them. Many properties are known about such domains, and a general purpose search method cannot hope to re-derive such properties in a feasible way. Moreover, there is a diverse and active community dedicated to finding efficient algorithms for computational problems in these domains. The question that must be addressed, and which is the focus of this position paper, is *how could such results be incorporated into the construction of powerful and practical constraint programming languages?* For logic programming to have the kind of impact that its advocates hope for, users must be able to "conveniently and efficiently" solve realistic large problems with logic programs. As for the dual goals of *convenience* (naturalness) and *efficiency*, note that the bulk of research in logic programming lies in logical analysis. This will surely contribute to the first goal, but only peripherally to the efficiency goal. It is well-known that the complexity of problems in the above classical domains must be carefully negotiated in order to elicit the tractable and avoid the intractable. Research in computational geometry and computer algebra has gained much insight into the boundary between the tractable and the intractable, and constraint programming can benefit by exploiting the fruits of this research.

The concern to incorporating powerful algebraic theories into the logic programs is addressed in the *Constraint Logic Programming* (CLP) framework of Jaffar and Lassez [16]. The fundamental issue of semantics and logical properties of such a framework has been worked out. What we hope to address are

*On sabbatical leave from Holy Cross College.

†Work supported by NSF grant #CCR-87-03458.

[1] Since it is hard to imagine logic without equality, these are usually identified as logical properties.

the algorithmic issues that arise, and how to incorporate powerful algorithmic techniques to the underlying interpreter/logic engine.

2 Geometric Domains

In this paper, we use the term "constraint programming" to refer to those aspects of logic programming concerned with incorporating various mathematical structures into logic languages.[2] Such aspects are dependent on the particular domain; we are not interested in "general truth maintenance". But "domain-dependence" is a relative concept as we wish to restrict our domain as little possible within certain (informal) parameters. Roughly speaking, the domains of interest in this paper can be described as "geometric". Algorithmic problems in such domains tend to have at most a single exponential (worst case) complexity (e.g., [6,10]). In contrast, domains which might be more "algebraico-geometric" tend to have double exponential complexity [21,9,33]. This contrast (which has been observed by researchers in computer algebra) is, informally, the difference between looking at complex varieties (geometry) as opposed to studying complex polynomial ideals (algebraic geometry). This observation suggests that we must exploit geometric properties in order to reduce complexity.

It might be objected that single-exponential complexity is still too high. In general, this is unavoidable [27,7] and a central goal of algorithmic research is to identify the interesting subcases which admit simpler solutions. This worst-case complexity can be ameliorated in several effective ways, including the use of randomization. The exponential behavior is often a function of only the dimensionality of a problem and with fixed dimensions, these problems becomes polynomial. In fact, much of the field of computational geometry concerns problems in 2 or 3 dimensions (this notion of dimensionality is to be distinguished from the dimensionality arising from the number of degrees of freedom). Experience has shown that although a domain may contain intractable problems, this worst-case behavior does not necessarily extend to all problems in the domain: it is often the case that problems of interest admit efficient solutions.

Finally, it must be argued that since the language PROLOG is already capable of simulating an alternating Turing machine [27], adding a constraint solver for a particular domain does not increase the asymptotic complexity of evaluating a constraint program. In fact, what we are proposing is the use of special purpose constraint solvers which can exploit the geometric aspects of the problem to speed the computation. One may wish to give up the expressive power of PROLOG and restrict oneself to a language with bounded complexity. But even here, most natural CLP languages have at least DEXP time query complexity [7,5].

We now classify the various geometric domains of interest.

One dimension. In "1-dimension", we have subrings of C: integers, rationals, real algebraic numbers and general algebraic numbers. All these domains have associated exact computational methods. Already here, one can run into (single) exponential time problems.

It should be emphasized that when we speak about integers and real numbers we do not mean "machine integers" or floating point numbers. While machine integers and floating point numbers are also appropriate for many problems, there are also many problem domains for which they are not well suited. Machine integers impose an *a priori* bound on the size of solutions (and intermediate results) that may be too restrictive for many problem domains. Floating point numbers provide an approximation of real values. This approximation includes a certain amount of error which unavoidable builts up as values are combined. The error which is accumulated while solving large systems may become unacceptably large. Furthermore, there are many problem domains (for example computational geometry) in which exact results are required. So one may think of "BigNum" exact arithmetic packages as the fundamental substitute for such machine numbers. We are currently working on several extensions to such packages such as the *variable precision* arithmetic which is described below.

General dimension. We can reduce a large part of geometric constraint programming to the solution of systems of algebraic equations. For our purposes, these equations have rational coefficients. The notion of "solutions" (cf. [34]) is by no means straightforward: the simplest is to indicate where the system is solvable. The next is to compute the dimension of the solution set. A more elaborate notion of solution is

[2] Admittedly, this terminology is far from perfect but it fits the general understanding of this term.

sometimes required: for instance, some applications need a sample point in each connected component of the solution set. The whole enterprise gets more complicated if we do not work in an algebraic closure. The most important instance of this is where we only want real solutions In the real field, we have the ordering property and the solution sets are *semi-algebraic* rather than algebraic. The complexity of these problems (algebraic and semi-algebraic) have been much clarified in recent years. *We shall think of these situations as setting the upper limit for what we hope to incorporate into a strong constraint logic language.* We note that the CLP($\mathcal{R}$) system [14] is targeted for exactly this domain. A major research goal is to identify subdomains where more efficient techniques can be used.

Linear and semilinear case. Mathematically, the linear case is essentially completely understood, being essentially linear algebra. But algorithmically, there are many unexplored questions. Consider a system of m linear equations in n variables, represented by an $m \times (n+1)$ matrix M. Note that all but the last column of M represents a variable. A fundamental question is how to determine maintain the solvability of the system M under insertion and deletion of equations? The complexity bound should be related to $L(M)$, the number of non-zero entries in M. Before considering the dynamic case, consider the following heuristic to determine the solvability of M: for each column of M that represents a variable that has only one non-zero entry, remove the row containing that entry. With each row removal, columns with only zeroes are created: these columns are also removed. Let M' be the reduced matrix. To justify this transformation, it is easy to show: *any solution of M' can be extended to a solution of M, and every solution of M can be obtained in this way.* (Of course, we should keep track of the deleted rows and columns so that we can recover the solution of M from the solution of M'.) We may now assume that every column in M' has at least two non-zero entries. We wish to repeat the above process. If a column has exactly two non-zero entries, we consider the two rows r, r' that contain these two entries: replace r' by a linear combination $r'' = ar + br'$, (a, b reals), so that the column now has only one non-zero entry. As above, we may delete r. So in effect, we have replaced r, r' by r''. Note that $L(M')$ is not increased in this way. Repeating this, we eventually obtain M'' in which every non-last column has at least 3 non-zero entries. In general, suppose every non-last column of M'' has at least $i \geq 3$ non-zero entries. We may continue this process of eliminating variables, but we can no longer ensure that $L(M'')$ is not increased. But there is a case where this still works: suppose $r_1, \ldots, r_i$ are i rows such that there are at least i columns (not counting the last column) all of whose non-zero entries all belong the these i rows. Moreover, the corresponding $i \times i$ submatrix is non-singular. Then by a linear transformation, we may convert the submatrix into triangular form. Then these i rows may be replaced by one of the transformed rows, justified as before, without increasing $L(M'')$. Although we cannot ensure this special case, this method might still work well in practice (compared to Gaussian elimination), a subject for experimental studies. The harder question is how to maintain the solvability above under insertion and deletion of equations. It seems that a randomized approach (see below) should work. Another instance of solving linear constraints is seen in [30,31].

By "semilinear" we refer to the use of linear inequalities. Much is known about this case via convex polytope theory and linear programming. The fundamental problem here is the computation of convex hulls. This is extremely well-studied in computational geometry and great strides have been made in recent years in understanding its computational complexity. In [19], a transformation is introduced which improves the detection of redundant linear inequalities. But the dynamic version of this problem (which is most critical for constraint programming) is only understood in the planar case. For instance, it is well-known that we can maintain the convex hull (=feasible region) of a set of halfspaces using $O(\log^2 n)$ time for insertion or deletion of a halfspace. Recently, Schwartzkopf [25,26] obtained efficient, apparently practical, randomized algorithms on maintaining convex hulls in arbitrary dimensions.

Planar and 3-D geometry. Applications of geometry in describing 2- or 3-D scenes is widespread, from CAD/CAM, to computer graphics, to physical simulations and robotics. In [11] we proposed a geometric editor called LINETOOL, based on polynomial constraints. Such an editor allows a user to construct a geometric scene using constraints and to query the scene (is this point inside this region?) and to revise the constraints. The underlying (numerical) representations is exact. Unfortunately, not much is known about how to exploit the special structure of planar constraints, and a general polynomial solver is still needed.

3 Polynomial Constraints

Most interesting constraint systems in the literature (other than purely logical constraints) can be modeled by systems of polynomial inequalities. There are several versions of this, and we briefly note the known algorithms and their prospects.

System of polynomial equations. The simplest constraint consists of a set of polynomials with rational coefficients. If we are interested in solutions in complex numbers then algorithms based on the Gröbner bases method ([2]) and Wu's method ([32,4]) are available. In fact, many computer algebra systems (MAPLE, MATHEMATICA, etc) has some version of the Gröbner basis algorithm. As an application for such systems, we can solve simple electrical circuits designs (cf. [29,15]). The worst case complexity here is essentially single exponential time.

Real solutions. If we are only interested in only real solutions to the above systems of equations, then the complete method is based on cell decompositions due to Collins. Here, the algorithm is essentially double exponential time. More recently, there are new methods (such as multivariate resultants) that avoid cell-decomposition and achieve single exponential space bounds [23,3,13]. In the real field, we have a total ordering so that we can allow inequalities. The solution sets are semialgebraic sets. For working over the reals, we can always replace inequalities by equalities if we are willing to introduce new variables: that is we replace an inequality $f \geq g$ by $f = g + a^2$ where a is a new real variable.

All of the above situations are infeasible in the worst case. In practice, Wu's method and Gröbner bases methods can be quite effective for some problems. In contrast, some newer resultant methods seems less useful because they achieve the worst case behavior for all input instances. Roughly speaking, the difference (between Wu's and Gröbner methods on one hand and the resultant methods on the other) is that the "best case behavior" of former is better than the best case behavior of the latter. But when we are interested in real solutions, all these methods yield only partial information, and we have nothing better than cell decomposition in some form. The real challenge is to exploit special properties of the problem at hand. For instance, these algorithms can be "dynamized" to allow users to add or delete polynomial constraints, while the system maintains a solution. This should be more efficient than resolving from scratch. In fact, many applications have only quadratic polynomials. Can this be exploited?

Zero-dimensions. One important case is the zero-dimensional case. In this setting some double-exponential time algorithms become single-exponential. We still get useful constraint systems based on 0-dimensional solvers: if the solution set has a positive dimension (dimensionality can be computed relatively fast) then the system is underspecified and we can ask the user to introduce more constraints. Alternatively (cf. [11]), we can ask the user to specify a subset of the variables as *independent*. If $U = u_1, u_2, \ldots, u_m$ are independent and $X = x_1, x_2, \ldots, x_n$ are dependent, we can treat the polynomial constraints as polynomials whose variables are X and whose coefficients belong to the field of rational functions of U. By choosing an appropriate subset of independent variables, we can produce a system whose solution set is zero-dimensional. The idea of dependent variables can be generalized into a dependency graph. The user can also be given an opportunity to reduce the dimensionality of the solution set by specifying additional constraints. For example, instead of identifying a variable as independent, the user could alternatively specify a value for that variable. Although the solution set produced will be less general, this has the advantage of being computationally faster. In this context, the work of Pedersen [22] on generalizing Sturm theory to higher dimensions is relevant. Like the classical Sturm theory, the generalized algorithm tells us about real solution points. Although this method is single-exponential time, its practical significance is not yet understood.

Variable clustering. In addition to the powerful methods which always succeed (but at high cost), we can hope to make large systems tractable by employing some heuristic methods. In particular, we can provide mechanisms for identifying clusters of variables that interact only minimally with other variables in the system. In the geometric editor LINETOOL[11], a user can usually identify such clusters. This is because we usually construct a cluster of geometric objects (points, lines, etc) relative to previously constructed objects. We could then solve the associated equations for these cluster variables, assuming that the "relatively independent" variables that they depend upon has been solved.

A notion related to variable clustering is the observation that many geometric objects introduces variables in groups. For instance, if we deal with planar point sets, then each point p introduces a pair of variables p_x, p_y. These pairs of variables are involved in constraints is a symmetric manner. For instance, the constraint that the distance between p and q is one becomes $(p_x - q_x)^2 + (p_y - q_y)^2 = 1$. Can such groupings be exploited? A recent paper of Stifter [28] indicates a promising approach.

Other techniques. A very potent technique to combat intractability is randomization. See the work of Kaltofen [17,18] and Schwartz [24]. Also, the use of datastructures in algebraic computing has not been developed up to this point, and we are investigating such issues.

4 Exact Arithmetic

In working with exact arithmetic over the integers or any algebraic number ring, the cost of performing arithmetic operations grows as the number of bits needed to represent the objects grow. Note that with algebraic numbers, "exact arithmetic" is possible although, beyond some approximation to the value, we must store auxiliary information such as the defining polynomial of the number.

There must be a clear understanding as to the fundamental importance of exact arithmetic. Constraint programming language implementations tend to use machine floating point arithmetic. Despite the great efficiency of machine arithmetic hardware, one is never sure of the result of such a computation. We could aim at a more modest goal of determining when the computed results are reliable. A well-known solution is to maintain intervals of uncertainty, but these intervals tend to grow so rapidly to as render correct computations suspect. At any rate, it is our belief that all such attempts to avoid exact computation is only a partial solution: no reliable constraint programming language implementation can avoid the ultimate use of some exact arithmetic package. On the other hand, we believe that the traditional BigNum packages must be re-engineered for use in logic programming languages. Recently, several promising approaches to doing exact arithmetic computation is explored in [12].

There are many situations in which approximate values with a pre-specified bounded error suffice for exact answers. In applications such as computer graphics, we may only need to locate a point up to the resolution of the screen. As a variation of this idea, in LINETOOL we allow arbitrary zooming, but here we may increase the resolution of a point as the application demand. Another application is checking inequality constraints. Sometimes, a low accuracy may be enough ascertain if a particular constraint is satisfied. Using approximate values can save much of the cost of computing with algebraic quantities. However, it is not clear how to determine *a priori* how much error can be tolerated in an expression. For example, if we wish to determine whether $a - b$ is greater then zero, we can tolerate coarse approximations of a and b, except in the case that the difference between the two values is small. To circumvent this problem, we propose looking at *variable precision* arithmetic. In this scheme, each value will be computed to a requested precision, but with the ability to refine the value to more precision if requested. Consider again the constraint $a - b > 0$. The values of a and b may be first computed to a coarse accuracy, perhaps within 0.0001. If the values are $a = 2.0123$ and $b = 1.9329$, then the constraint is satisfied. But, if instead the values at this accuracy were determined to be $a = 2.0123$ and $b = 2.0122$, then more precision is needed and requested. In any event, we will determine with no uncertainty whether the constraint is satisfied.

The solution values which satisfy the constraint system may themselves be computed only to within acceptable tolerances. In this case, requests for enhanced precision may come directly from the user. The system can include a value *prober* to allow the user to directly query result values and determine properties satisfied by these values.

5 Final Remarks

Incorporating geometric constraints into logic programming requires sophisticated algorithmic techniques because a general purpose solver would generally be too inefficient. Many new algorithmic issues arise. To test some of these ideas, we are building a prototype of a planar geometry editing system in the style of ThingLab or LINETOOL. One goal of this system is to produce a topologically correct model, which can accurately represent the relationships between points which may be arbitrarily close together. To achieve

this goal, we use use exact methods and algebraic numbers. The high cost associated with algebraic numbers, is somewhat alleviated by the use of variable precision techniques which do not sacrifice the exactness of the computation. Nothing in the constraint solving and exact model representation are specific to the domain of planar geometry. With some modifications, this subsystem could be reused for other applications. We believe that a carefully engineered subsystem dealing with linear constraints is already very useful (see [31,19]), presenting many interesting research question.

References

[1] A. Borning. The programming language aspects of thinglab, a constraint-based simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3:353–387, 1981.

[2] B. Buchberger. History and basic features of the critical-pair/completion procedure. *J. Symb. Comput.*, 3:3–38, 1987.

[3] J. Canny. Some algebraic and geometric computations in pspace. In *Proceedings of the Twentieth Annual ACM Smposium on Theory of Computing*, pages 460–467, 1988.

[4] S.C. Chou. Proving elementary geometry theorems using Wu's algorithm. *Contemporary Mathematics*, 29:243–286, 1984.

[5] J. Cox and K. McAloon. *Decision Procedures for Constraint Based Extensions of Datalog*, chapter 2. MIT-Press, 1993.

[6] J. Cox, K. McAloon, and C. Tretkoff. Computational complexity and constraint logic programming languages. In *Logic Programming. Proceedings of the North American Conference*, pages 401–415, 1990.

[7] J. Cox, K. McAloon, and C. Tretkoff. Computational complexity and constraint logic programming languages. *Annals of Math. and Artificial Intellegence*, 5:163–190, 1992.

[8] D. DeGroot and G. Lindstrom. *Logic Programming. Functions, Relations, and Equations*. Prentice-Hall, 1986.

[9] Thomas W. Dubé. The structure of polynomial ideals and Gröbner bases. *SIAM Journal on Computing*, 19(4):750–773, 1990.

[10] Thomas W. Dubé. A combinatorial proof of the effective nullstellensatz. *J. Symb. Comput.*, 1993. to appear.

[11] Lars Warren Ericson and Chee K. Yap. The design of LINETOOL: a geometric editor. *ACM Symposium on Computational Geometry*, 4:83–92, 1988.

[12] Steven Fortune and Christopher van Wyk. Efficient exact arithmetic for computational geometry. *Symposium on Computational Geometry*, 9, 1993. To appear.

[13] D. Grigor'ev and N. Vorobjov. Solving systems of polynomial inequalities in subexponential time. *Journal of Symbolic Computation*, 5:37–64, 1988.

[14] N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. *The CLP(R) Programmer's Manual*, version 1.2 edition, September 1992.

[15] N. Heintze, S. Michaylov, and P. Stuckey. Clp(r) and some electrial engineering problems. In J.-L. Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 675–703, 1987.

[16] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[17] E. Kaltofen. Factorization of polynomials given by straight-line programs. In S. Micali, editor, *Randomness in Computation: Advances in Computing Research*. JAI Press, Greenwich, Connecticut, 1987.

[18] E. Kaltofen. Greatest common divisors of polynomials given by straight-line programs. *Journal of the ACM*, 35:231–264, 1988.

[19] J.L. Lassez, T. Huynh, and K. McAloon. Simplification and elimination of redundant linaer arithmetic constraints. In *Logic Programming, Proceedings of the North American Conference*, pages 37–51, 1989.

[20] William Leler. *Constraint Programming Languages: their specification and generation*. Addison-Wesley, Reading, Massachusetts, 1988.

[21] Ernst W. Mayr and Albert R. Meyer. The complexity of the word problems for commutative semigroups and polynomial ideals. *Advances in Mathematics*, 46:305–329, 1982.

[22] Paul Pedersen. Counting real zeroes. Technical Report 243, NYU-Courant Institute, Robotics Laboratory, 1990. Courant Institute doctoral thesis.

[23] J. Renegar. On the computaional complexity of approximating solutions for real algebraic formulae. Technical Report 858, School of Operations Research and Industrial Engineering, College of Engineering, Cornell University, 1989.

[24] Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27:701–717, 1980.

[25] Otfried Schwarzkopf. Dynamic maintenance of geometric structures made easy. *IEEE Symp. on Foundations on Computer Science*, 32:197–206, 1991.

[26] Otfried Schwarzkopf. *Dynamic maintenance of convex polytopes and related structures*. PhD thesis, Department of Mathematics, Free University Berlin, 1992.

[27] Ehud Shapiro. Alternation and the computational complexity of logic programs. *J. Logic programming*. 1:19–33, 1094.

[28] Sabine Stifter. Geometry theorem proving in vector spaces by means of Gröbner bases. RISC-Linz Report Series No. 93-12, Research Inst. for Symbolic Comp., Johannes Kepler Univ., Austria, 1993.

[29] G. Sussman and G. Steele. Constraints – a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[30] Christopher J. van Wyk. A high-level language for specifying pictures. *ACM Trans. on Graphics*, 1:163–182, 1982.

[31] Christopher J. van Wyk. Arithmetic equality constraints as C++ statements. *Software-Practice and Experience*, 0:1–27, 1991.

[32] W.T. Wu. Basic principles of mechanical theorem proving in geometries. *Journal of Automated Reasoning*. 2:221–252, 1986.

[33] Chee-Keng Yap. A new lower bound construction for commutative Thue systems with applications. *Journal of Symbolic Computation*, 12:1–28, 1991.

[34] Chee-Keng Yap. *Fundamental Problems in Algorithmic Algebra*. Princeton University Press, (to appear, 1994).

On the Semantics of Optimization Predicates in CLP languages

François Fages

LIENS-CNRS, Ecole Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France
fages@dmi.ens.fr

and

LCR Thomson-CSF, Domaine de Corbeville, 91404 Orsay Cedex, France.

Abstract

The Constraint Logic Programming systems which have been implemented include various higher-order predicates for optimization. In CLP(FD) systems, several optimization predicates, such as `minimize(G(X),f(X))`, `minimize-maximum(G(X),[f1(X),...,fn(X)])`, are implemented by using branch and bound algorithms. In CLP(R) systems, the Simplex algorithm used for satisfiability checks can also be used for linear optimization through the predicate `rmin(f(X))` which adds to the constraints on X the ones defining the space where the linear term $f(X)$ is minimized. These optimization constructs do not belong however to the formal CLP scheme of Jaffar and Lassez, and they lack a declarative semantics. In this paper we propose a general definition for optimization predicates, for which one can provide both a logical and a fixpoint semantics based on Kunen-Fitting's semantics of negation. We show that the branch and bound algorithm can be derived as a refinement of the implementation of the semantics using CSLDNF-resolution, and that the branch and bound algorithm can be lifted to a full first-order setting with constructive negation.

1 Introduction

The Constraint Logic Programming systems which have been implemented include various higher-order predicates for optimization. In CLP(FD) systems as CHIP, defined over finite domains, several optimization predicates, such as `minimize(G(X),f(X))`, or `minimize-maximum(G(X),[f1(X),...,fn(X)])`, are implemented with branch and bound algorithms [8]. In CLP(R) systems, defined over real numbers, the Simplex algorithm used for satisfiability checks can also be used for linear optimization through the predicate `rmin(f(X))` which adds to the constraints on X the ones defining the space where the linear term $f(X)$ is minimized. These optimization constructs do not belong however to the formal CLP scheme of Jaffar and Lassez [5], and they lack a declarative semantics.

The first problem to solve is the dependence of the result on the ordering of the goals. In many systems indeed the constraints on the variables appearing in the optimization goal are passed to the optimization process, producing the following problematical behavior:

```
p(X) :- X>=0.

? X>=1 , minimize(p(X),X).
X=1

? minimize(p(X),X) , X>=1.
no
```

Clearly the optimization process should be localized to the goal given as argument, and the other constraints inherited from the other goals should not change the optimality condition. Therefore the correct answer in the previous example is *no*. If $X = 1$ was the intended answer, one should write:

```
? minimize((X>=1,p(X)),X).
X=1
```

With this provision one can give a declarative reading to CLP programs containing optimization predicates. We show that Kunen's three-valued semantics of logic programs with negation is all we need to do so, and that optimization predicates can thus be treated as higher-order constraints. After reviewing completeness results in the CLP scheme, we show that the well-known branch and bound algorithm can be derived as a refinement of the implementation of the semantics using CSLDNF-resolution, and that the branch and bound algorithm can be lifted to a full first-order setting with constructive negation.

2 The declarative semantics of optimization predicates

Definition 1 *Let $(A, \leq)$ be a totally ordered structure. The* minimization *higher-order predicate*

$$min(G(X,Y),[X],f(X,Y))$$

is defined as a notation for the formula:

$$G(X,Y) \wedge \forall Z\ (G(X,Z) \supset f(X,Z) \not< f(X,Y))$$

An optimization constraint logic program (OCLP) (resp. goal) is a CLP program (resp. goal) which may contain occurrences of the minimization predicate in rules bodies (resp. in goals).

The second argument to the predicate *min* is a possibly empty list of "protected" variables, $[X]$. Only the variables of the goal, away from X, are affected by the optimality condition.

As is well known, general first-order formulas can be normalized [6]. An OCLP program P can be transformed into an equivalent normal CLP program containing negations, by replacing each occurrence of the atom

$$min(G(X,Y),[X],f(X,Y))$$

by the conjunction of atoms

$$G(X,Y), \neg p(X,Y)$$

where p is a new predicate symbol, and by adding to the program the rule:

$$p(X,Y) \leftarrow f(X,Z) < f(X,Y) | G(X,Z).$$

In the following, $\overline{P}$ denotes the normal CLP program obtained by repeatedly applying this transformation to P, and P^* denotes the Clark's completion [6] of P (without Clark's equality axioms as symbols are interpreted in A).

Definition 2 *The semantics of an OCLP program P is the set of 3-valued consequences of $\overline{P}^* \wedge th(A)$. A correct answer to an OCLP query G and a program P is a set of constraints c such that*

$$\overline{P}^* \wedge th(A) \models_3 \forall(c \rightarrow G) \wedge \exists(c)$$

Going back to the example of the introduction, we can check that the correct answer to

$$X \geq 1, min(p(X),[\,],X)$$

is *no*, that $X \geq 1$ is a correct answer to

$$X \geq 1, min(p(X),[X],X),$$

and that $X = 1$ is the correct answer to the goal

$$min(X \geq 1 | p(X),[\,],X).$$

In general the answers of an OCLP query are non ground, and can be arbitrary sets of constraints. Considering the rule

$$p(X,Y) \leftarrow 0 \leq X, X \leq Y.$$

$X = Y$ is a correct answer to the query

$$min(p(X,Y),[\,],Y-X).$$

The definition of OCLP programs does not exclude recursion though optimization predicates. A similar problem is discussed in [1] and [2] where the stratified semantics of aggregates are generalized using the well-founded and stable model semantics of logic programs. In the context of OCLP programs, the natural theory to apply is Fitting-Kunen's 3-valued semantics, which does not coincide with stratified and well-founded semantics. ¿From a practical point of view one can notice that definite OCLP programs usually don't contain recursion through optimization. In general however, we have:

Proposition 1 *Any normal logic program is equivalent to a definite OCLP program.*

Proof: Let us consider the OCLP program over the natural numbers obtained from the normal logic program by replacing each negative literal $\neg p(X)$ by $max(q(X,y),[X],y)$ where q is a new predicate symbol, and by adding the rules

$$q(X,0).$$

$$q(X,y) \leftarrow p(X).$$

We have $\exists X \exists y\ max(q(X,y),[X],y)$ iff $\exists X \exists y \forall z\ q(X,y) \wedge \neg(q(X,z) \wedge z > y)$
iff $\exists X \exists y\ (y = 0 \wedge \neg p(X)) \vee (p(X) \wedge \forall z \neg(\alpha(X,z) \wedge z > y))$
iff $\exists X \exists y\ y = 0 \wedge \neg p(X)$
iff $\exists X \neg p(X)$. □

3 Completeness results

The completeness result of SLDNF-resolution w.r.t. to the three-valued semantics of logic programs [4] relies on the properties of the finite powers of Fitting's operator Φ_P^A. These properties generalize to normal CLP programs:

Theorem 1 *[3] [7] Let A be a structure and P a normal CLP program. Then the following are equivalent:*

- $\Phi_P^A \uparrow n(c|G) = t$ *for some finite* n,
- $th(A) \wedge \overline{P}^* \models_3 \forall(c \rightarrow G) \wedge \exists(c)$

In this way Stuckey [7] proved the completeness of CSLDCN-resolution (i.e. constraint SLD-resolution with constructive negation) for normal CLP programs. CSLDCN-resolution is based on the CSLD inference rule for positive goals, and on the CSLDCN inference rule for negative goals:

$$CSLD \frac{\leftarrow c|A_1,...,A_i,...,A_n \quad th(A) \models \exists(c')}{\leftarrow c'|A_1,...,A_{i-1},B_1,...,B_m,A_{i+1},...,A_n}$$

where $c' = c \wedge A_i = B$, $(B \leftarrow B_1,...,B_m) \in P$.

$$CSLDCN \frac{\leftarrow c|A_1,...,\neg A_i,...,A_n \quad th(A) \models \exists(c')}{\leftarrow c'|A_1,...,A_{i-1},A_{i+1},...,A_n}$$

where $c' = c \wedge \neg\exists^{\sim}(c_1) \wedge ... \wedge \neg\exists^{\sim}(c_n)$ (the existential closure being over variables not in c),
and $\{c \wedge c_1,...,c \wedge c_n\}$ are the successful derivations of the goal $\leftarrow c \wedge A_i$.

Theorem 2 *[7] Let P be a normal OCLP program over a structure A.*

If $th(A) \wedge \overline{P}^* \models_3 \forall(c \rightarrow G) \wedge \exists(c)$ then the CSLDCN-derivation tree for $(c|G)$ contains successful derivations with constraints $c_1,...,c_n$, such that $A \models c \supset \exists^{\sim}c_1 \vee ... \vee \exists^{\sim}c_n$.

If $th(A) \wedge \overline{P}^* \models_3 \forall(c \rightarrow \neg G) \wedge \exists(c)$ then the CSLDCN-derivation tree for $(c|G)$ is finitely failed.

In particular the completeness of CSLDNF-resolution (i.e. constraint SLD-resolution with negation by failure) follows under the non-floundering assumption.

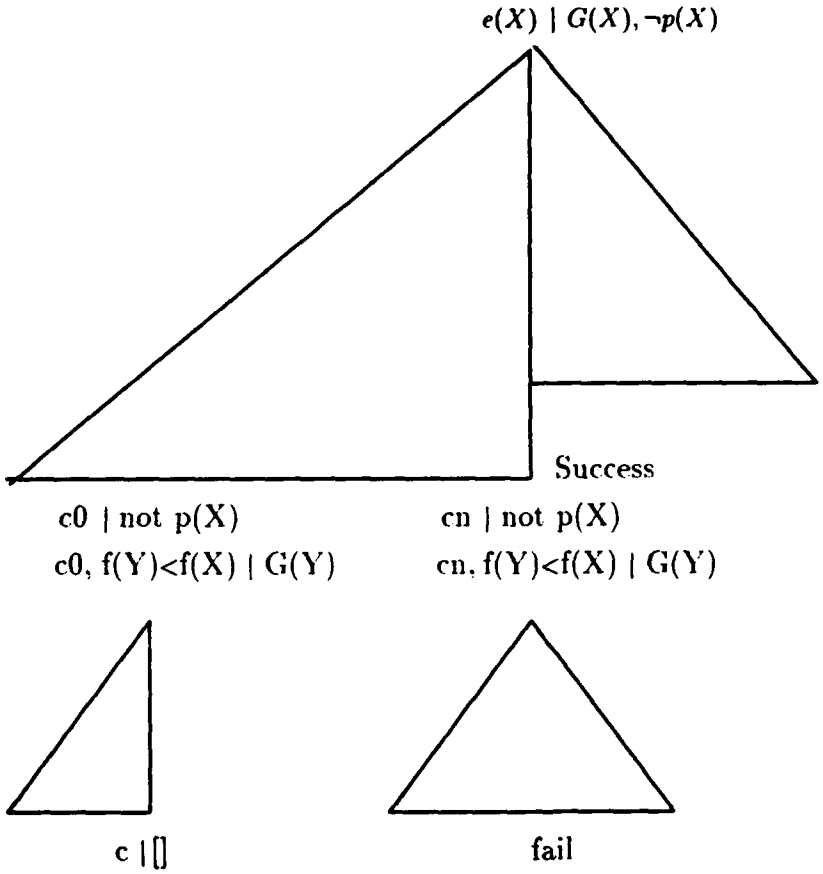
4 The branch and bound algorithm as negation by failure

In this section we study the application of the optimization predicate over a goal $G(X,Y)$, such that all the successful CSLD-derivations of $G(X,Y)$ instanciate the arguments X and Y to some values. This is typically the case of optimization in CLP(FD), where enumeration is mixed with constraint propagation in order to palliates the incompleteness of the constraint solvers [8].

Under these assumptions, it is clear that the negative goals introduced by the optimization predicates (in $G(X,Y), \neg p(X,Y)$) never flounder. Thus CSLDNF-resolution is complete w.r.t. the (3-valued) semantics of such OCLP programs. For simplicity, let us consider the goal

$$e(X)|min(G(X),[\,],f(X)),$$

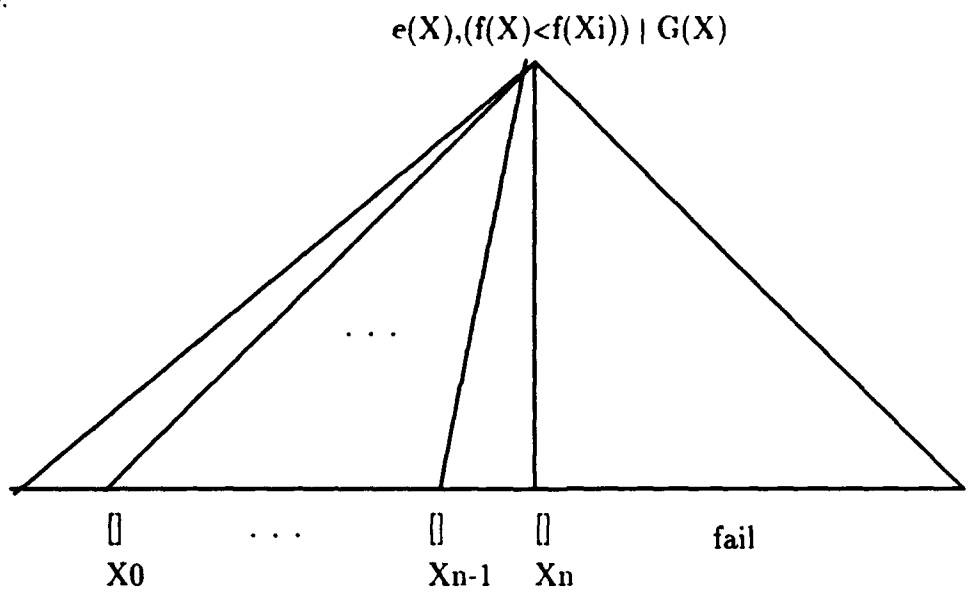
and its CSLDNF-derivation tree.



For the moment let us consider the search for only one successful derivation of the optimization goal, not all successful derivations. Under the normal left-right order traversal of the tree, when a successful derivation is obtained for $e(X)|G(X)$ with constraint c_i, then $c_i|\neg p(X)$ remains to be shown, and for this, another derivation tree is developed for the goal $c_i, f(Y) < f(X)|G(Y)$. If this subtree is finitely failed then we obtain a successful derivation for the optimization goal. Otherwise if the derivation subtree contains a successful derivation, then it is a failure for the optimization goal, and the successful subderivation is lost. Therefore a derivation subtree for G is developed for each successful derivation of $e(X)|G(X)$.

The well-known branch and bound algorithms can be presented as optimized versions of CSLDNF-resolution procedures, that exploit the successful derivations found in the refutation of the optimality of a solution. In the *backtracking version* of the branch and bound algorithm (BB), a *single* derivation tree for G is developed. When a successful derivation is found (under the left-right order traversal), the corresponding

solution X_i is memorized, and the search by backtracking continues with the additional constraint $f(X) < f(X_i)$. The additional constraint is used to prune the search space and explore only a portion of the derivation tree:



In the BB algorithm, the last memorized solution, X_n, is a solution to:

$$min(e(X)|G(X), [\,], f(X))$$

To show that X_n is indeed a solution to

$$e(X)|min(G(X), [\,], f(X))$$

it suffices to check that the goal

$$e(X), f(X) < f(X_n)|G(X)$$

fails. If it is not the case, then the goal $e(X)|min(G(X), [\,], f(X))$ must fail. The gain in efficiency over CSLDNF-resolution is obvious as only two derivation trees are thus developed in this way.

Therefore two algorithms are possible for finding the answers to the goal

$$e(X)|min(G(X), [\,], f(X))$$

depending whether the branch and bound algorithm is applied initially to $e(X)|G(X)$ or to $G(X)$:

Algorithm 1 *BB algorithm with environment constraints.*

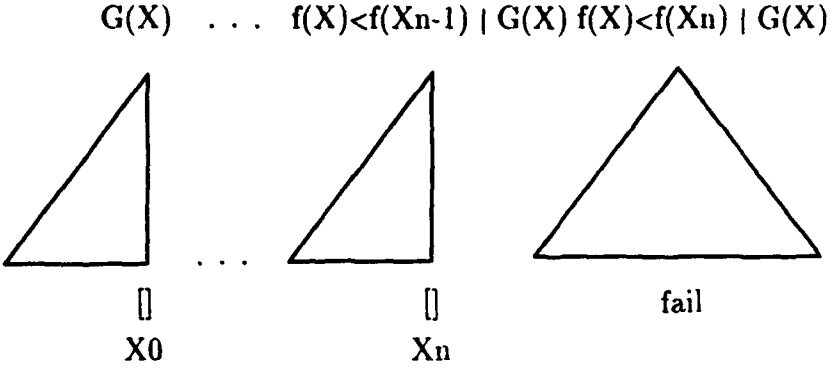
1. *compute one solution X_n to $min(e(X)|G(X), [\,], f(X))$ by using BB algorithm,*
2. *check by CSLDNF-resolution that $e(X), f(X) < f(X_n)|G(X)$ admits no solution, otherwise fail,*
3. *return X_n, or if all solutions are needed, return the answers to $e(X), f(X) = f(X_n)|G(X)$ computed by CSLDNF-resolution.*

Algorithm 2 *BB algorithm without environment constraints.*

1. *compute one solution X_n to $min(G(X), [\,], f(X))$ by using BB algorithm,*
2. *return the solutions to $e(X), f(X) = f(X_n)|G(X)$ computed by CSLDNF-resolution.*

In actual CLP(FD) systems with optimization predicates, algorithm 1 without step 2 is generally implemented, hence the difficulties mentioned in the introduction concerning the declarative semantics and the possibility to treat optimization predicates as higher-order constraints. Algorithm 2 does not use the constraints inherited from the environment to prune the search space for finding the optimal cost of a solution (step 1). Note however that, under termination assumptions, step 1 in algorithm 2 can be done at compile time.

Note also that other versions than the backtracking version of the branch and bound algorithm can be preferred for implementation in a CLP system. In the *iterative version* of the branch and bound algorithm, once a successful derivation for $G(X)$ is found, the corresponding solution X_0 is memorized, and another derivation tree is developed for $f(X) < f(X_0) \mid G(X)$. When the derivation tree is finitely failed, the last memorized solution is optimal.



When heuristic search techniques are used in combination with constraint propagation, the iterative version of the branch and bound algorithm makes it possible to change the order in which goals are selected, according to the new constraints added and to the heuristic. For this reason the iterative version can be practically more efficient.

5 The branch and bound algorithm lifted to the full first-order setting with constructive negation

CSDLCN-resolution provides a complete procedure for general OCLP programs without the non-floundering assumption. Let us consider the goal $min(G(X), [], f(X))$. On a successful derivation of $G(X)$ with constraint $c_i(X)$, constructive negation for the remaining goal

$$c_i(X) \mid \neg p(X)$$

consists in developing a complete derivation tree for

$$c_i(X), f(Y) < f(X) \mid G(Y)$$

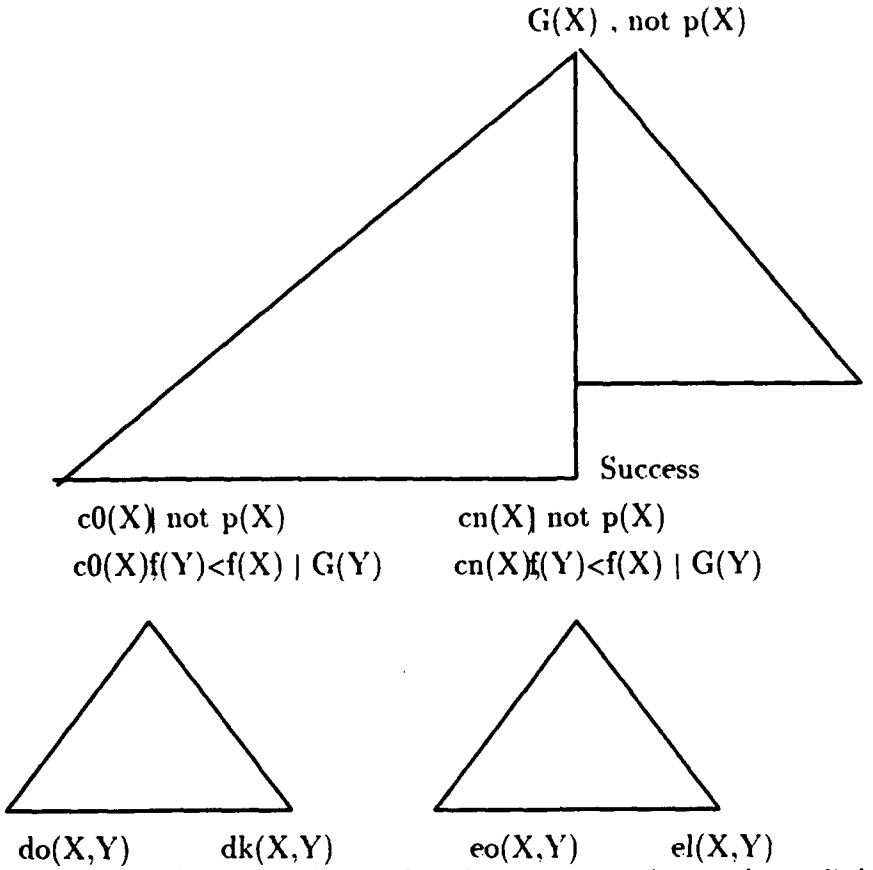
If $c_i(X) \wedge d_0(X,Y), ..., c_i(X) \wedge d_k(X,Y)$ are the constraints associated to the successful derivations of this tree, then the negative goal is successful if the constraint

$$\forall Y \; c_i(X) \wedge \neg d_0(X,Y) \wedge ... \wedge \neg d_k(X,Y)$$

is satisfiable[1].

Therefore a complete derivation tree for G is developed for each successful derivation of $G(X)$ not satisfying that condition:

[1] Note that if the structure is admissible [7] this condition is equivalent to a conjunction of existentially quantified disjunctions of conjunctions of admissible constraints.



Now the transformations described in the previous section can be applied in a similar fashion here in order to generalize the branch and bound algorithm to a full first-order setting. For instance the iterative version of the generalized branch and bound algorithm consists in finding a successful derivation for $G(X)$, say with constraint $c_0(X)$. then iterate finding a successful derivation for the goal

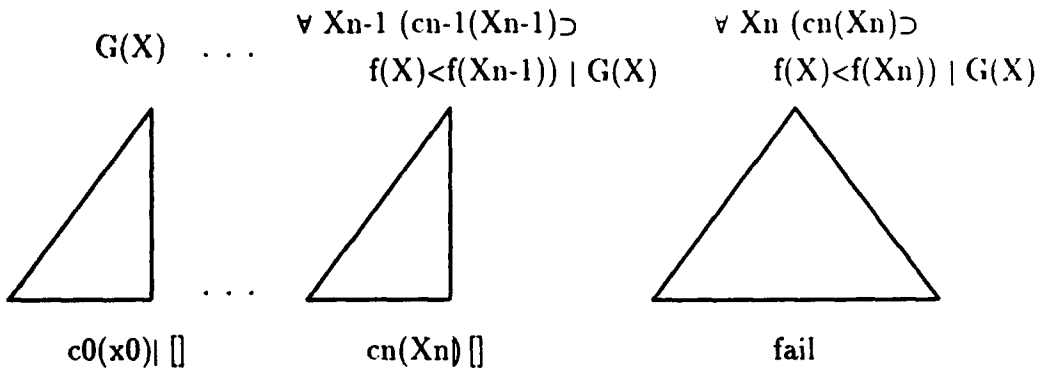
$$\neg\exists X_0(c_0(X_0) \wedge f(X_0) \leq f(X))|G(X)$$

which is equivalent to

$$(\forall X_0\ c_0(X_0) \supset f(X) < f(X_0))|G(X).$$

Note that as the structure is a total order, the constraint $(\forall X_0\ c_0(X_0) \supset f(X) < f(X_0))$ is equivalent to the constraint without universal quantifier $f(X) < k_0$ where $k_0 = min_{c_0(X_0)} f(X_0)$, when it exists. In particular in CLP(R), linear programming algorithms permit to decide efficiently the constraints involved in that restricted form of constructive negation, without having to rely on the admissibility of the structure R result [7] which is based on generally unpractical quantifier elimination techniques.

The derivation trees developed in the iterative first-order branch and bound procedure are thus the followings:



The procedure stops when the derivation tree is finitely failed, in which case the last memorized solution, say $c_i(X)$, is such that

$$\overline{P}^* \wedge th(A) \models_3 \forall X \; c_i(X) \supset G(X)$$

$$\overline{P}^* \wedge th(A) \models_3 \neg\exists Y(\forall X \; c_i(X) \supset f(Y) < f(X))|G(Y)$$

that is

$$\overline{P}^* \wedge th(A) \models_3 \forall Y G(Y) \supset (\exists X \; c_i(X) \wedge f(Y) \not< f(X))$$

hence

$$c_i(X) \wedge \forall Y \neg(c_i(Y) \wedge f(Y) < f(X))$$

is satisfiable, and is an optimal solution.

In this way both algorithms 1 and 2 of the previous section can be generalized to a full first-order setting:

Algorithm 3 *CLP-BB procedure with environment constraints.*

1. *compute one answer constraint* $c_n(X)$, *to* $min(e(X)|G(X), [\,], f(X))$ *by using BB algorithm,*
2. *check by CSLDNF-resolution that* $e(X), c_n(X_n), f(X) < f(X_n)|G(X)$ *admits no successful derivation, otherwise fail,*
3. *return* $c_n(X)$, *or if all solutions are needed, return the answers to* $e(X), c_n(X_n), f(X) = f(X_n)|G(X)$ *computed by CSLDNF-resolution.*

Algorithm 4 *CLP-BB procedure without environment constraints.*

1. *compute one answer constraint* $c_n(X)$ *to* $min(G(X), [\,], f(X))$ *by using BB algorithm,*
2. *return the solutions to* $e(X), c_n(X_n), f(X) = f(X_n)|G(X)$ *computed by CSLDNF-resolution.*

6 Conclusion

Optimization higher-order predicates in CLP systems can be given a logical semantics based on the three-valued consequences of logic programs with negation. We have shown that the well-known branch and bound algorithms can be presented in this framework as specific optimizations of CSLDNF-resolution procedures. Applying the same optimizations to CSLDCN-resolution, which is based on constructive negation, we obtained a powerful generalization of the branch and bound algorithms to a full first-order setting, including linear programming as a deterministic particular case.

Acknowledgement

It is a pleasure to acknowledge fruitful discussions with my colleagues at LCR and at LIENS, and to thank Peter Stuckey for providing me with the references to the results on aggregates.

References

[1] S. Ganguly, S. Greco, C. Zaniolo, "Minimum and maximum predicates in logic programming". Proc. of PODS'91, Denver, pp. 154-163. 1991.

[2] D.B. Kemp, P.J. Stuckey, "Semantics of logic programs with aggregates", Proc. of ILPS'91, San Diego, pp.387-401. 1991.

[3] K. Kunen, "Negation in logic programming", Journal of Logic Programming, 4(3), pp.289-308, 1987.

[4] K. Kunen, "Signed data dependencies in logic programming", Journal of Logic Programming, 7(3), pp.231-245, 1989.

[5] J. Jaffar, J.L. Lassez, "Constraint Logic Programming", Proc. of POPL'87, Munich. 1987.

[6] J.W. Lloyd, "Foundations of Logic Programming", Springer Verlag. 1987.

[7] P. Stuckey, "Constructive negation for constraint logic programming", Proc. LICS'91, 1991.

[8] P. Van Hentenryck : "Constraint Satisfaction in Logic Programming", MIT Press 1989.

A higher-order extension of constraint programming in discourse analysis

Tim Fernando
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
fernando@cwi.nl

Abstract

Variables on which constraints are imposed incrementally can be said to carry "existential" force in the following sense. Under a translation, commonly used in analyzing natural language discourse, of first-order formulas into programs from quantified dynamic logic, such a variable is introduced (at the level of formulas) by an existential quantifier. The present paper extends that translation to support constraints on variables introduced, as it were, by universal quantification. The extension rests on a certain program construct $\Rightarrow$ that can be interpreted (following Kleene's realizability analysis of universal-existential clauses) by closing the collection of states on which the programs act under partial functions. An alternative "reduced" interpretation of $\Rightarrow$ can also be given over sets of states from dynamic logic (not unlike Concurrent Dynamic Logic). These interpretations can be related by what is essentially a reduction to disjunctive normal form, involving so-called and-or computations.

A formalism for analyzing natural language discourse that has received considerable attention in certain linguistic circles is *Discourse Representation Theory* (DRT), due to Kamp [8] and Heim [6]. The basic idea in DRT is to capture the information a piece of discourse contributes by interpreting a natural language sentence semantically as a binary relation on so-called *Discourse Representation Structures* (DRS's). A DRS is a pair (D, C) consisting of a set D of *discourse markers* and a set C of *conditions* on them, formulated as first-order formulas whose free variables are among the discourse markers. For example, the piece of discourse

A man walks.

induces a transition from the empty DRS $(\emptyset, \emptyset)$ to the DRS $(\{x\}, \{\mathtt{man}(x), \mathtt{walk}(x)\})$. Similarly,

He sees a house.

sends $(\{x\}, \{\mathtt{man}(x), \mathtt{walk}(x)\})$ to $(\{x, y\}, \{\mathtt{man}(x), \mathtt{walk}(x), \mathtt{house}(y), \mathtt{see}(x, y)\})$. It follows by relational composition that

(*) A man walks. He sees a house.

takes $(\emptyset, \emptyset)$ to $(\{x, y\}, \{\mathtt{man}(x), \mathtt{walk}(x), \mathtt{house}(y), \mathtt{see}(x, y)\})$. As will be made precise below, conditions are introduced in a monotonic manner, with attention to consistency and entailment. That is, a condition amounts to what is called a "constraint" in constraint programming. Thus, it is not unreasonable to describe DRT as a form of constraint programming for discourse analysis. The same description applies to an extension of DRT presented below,[1] the purpose of which is best described by illustration. Notice that the variable for

[1] Readers familiar with Saraswat [14], however, should be cautioned that in the present paper, (i) quantifiers are employed to introduce variables on which constraints can then be imposed, and (ii) parallel computations arise when a process spawns multiple processes, which then proceed in a "conjunctive" fashion, with identically initialized but separate stores. This contrasts with Saraswat [14], where an existential quantifier is used to "hide" a variable, which, otherwise, might be shared by processes running in parallel. (The concept of locality or scope analyzed in Saraswat [14] should not be confused with what is called "force" below.) *No* claim is (of course) made here that (i) and (ii) represent the "proper" notions of quantification and parallelism; only that there are works (e.g., Groenendijk and Stokhof [4], Peleg [13], and the references cited therein) where notions of quantification and parallelism different (in essential ways) from Saraswat [14] are studied (and that such notions are what concern the present paper). The clash in terminology is unfortunate, and surely ought to be resolved. The concluding discussion returns to this point.

the **man** in (*) is implicitly bound by an existential quantifier. (This implicit binding is made explicit in a translation discussed below, under which **a man** is rendered as $\exists x$ **man**(x).) By contrast the variable for the **player** in

(**) **Every player chooses a pawn. He puts it on square one.**

(from Groenendijk and Stokhof [4]) is bound universally. Now, the object of the present extension is

(†) to support the imposition of conditions on variables with *universal* force, under a framework where conditions combine by relational composition.

That is, the challenge of (†) is to analyze (**) under compositional principles identical to that for

A certain player chooses a pawn. He puts it on square one.

which is analyzed in much the same way as (*). Briefly, the difficulty is that the pronoun **he** in (**) must refer not only to a particular **player**, but to every **player**. The solution below was introduced in Fernando [2], where applications to natural language are taken up. The point of the present paper (beyond relating the work mentioned to constraint programming) is to examine the rather remarkable feature of the extension that a notion of parallel computation is introduced via an old "constructive" idea involving higher-order witnesses.

To describe DRT formally, it is useful to start with first-order formulas (to which a translation from certain natural language utterances is taken for granted), and to characterize DRT as a function from first-order formulas to *meanings*. To bring out the semantic (as opposed to syntactic) character of these meanings, DRT can be presented through a slight variant, Groenendijk and Stokhof [4]'s *Dynamic Predicate Logic* (DPL), which is, in turn, based on quantified dynamic logic (see, for example, Harel [5]). DPL translates first-order formulas A to programs A^{DPL} from dynamic logic as follows[2]

$$\begin{aligned} A^{\mathsf{DPL}} &= A? \quad \text{for atomic } A \\ (A\&B)^{\mathsf{DPL}} &= A^{\mathsf{DPL}} \,;\, B^{\mathsf{DPL}} \\ (\exists x\, A)^{\mathsf{DPL}} &= x :=? \,;\, A^{\mathsf{DPL}} \\ (\neg A)^{\mathsf{DPL}} &= \neg\,(A^{\mathsf{DPL}})\,, \end{aligned}$$

where the negation $\neg p$ of a program p is the dynamic logic test checking that p cannot terminate. The customary notation

$$([p]\ \bot)\ ?$$

exposes two properties of this form of negation that are not intrinsic to the concept of negation — viz., its *universal* character (reflected by the square brackets in the modality $[p]$) and its *static* character (reflected by ? — recall that a dynamic logic test $A?$ cannot return an output state distinct from its input state, and is, in this sense, static). The other connectives are derived as in classical logic, e.g.,

$$\begin{aligned} \forall x\, A &= \neg\exists x\, \neg A \\ A \supset B &= \neg(A\&\neg B) \end{aligned}$$

and accordingly inherit the universal and static properties of negation. Whereas some notion of universality is essential to universal quantification and implication, the goal (†) above requires *dynamic* (as opposed to static) forms of universal quantification and implication (so that, for instance in (**), the **player** introduced by the first sentence can serve as the referent for **he** in the second sentence). It is simple enough to introduce an alternative form $\sim$ of negation that is neither universal nor static, by defining $\sim$ as a map on a richer

[2]The semantic interpretation of programs in dynamic logic is reviewed in section 1.1. The non-commutative treatment of conjunction & below may seem odd to readers unfamiliar with discourse analysis. The intention is to capture the sequentiality in processing discourse. The reader can, if she prefers, write $A.B$ instead of $A\&B$.

collection Φ of formulas obtained by closing predicate symbols under so-called anti-extensions, and treating $\forall, \supset$ and $\vee$ as primitive, thereby allowing De Morgan's laws to be incorporated into $\sim$. (The map $\sim$ goes back at least to Nelson [11], and has come to be known in the literature as *strong negation*.) What requires a bit more work is interpreting universal quantification and implication dynamically. That interpretation is most conveniently presented by a translation $\cdot^e$ of the richer collection Φ of formulas into a collection P of programs that is correspondingly richer than the programs of dynamic logic. The crucial clauses of $\cdot^e$, when compared to $\cdot^{\mathsf{DPL}}$, are

$$\begin{aligned}(A \vee B)^e &= A^e + B^e \\ (A \supset B)^e &= A^e \Rightarrow B^e \\ (\forall x\ A)^e &= (x :=?) \Rightarrow A^e\ .\end{aligned}$$

While programs in dynamic logic are closed under non-deterministic choice $+$, the program construct $\Rightarrow$ gives rise to new programs, requiring, in fact, a higher-order extension of dynamic logic's states. That extension can take the form either of

(i) closing the collection of states inductively under a partial function construct, whereupon the interpretation of implication and universal quantification in Kleene [9] can be adapted to $\Rightarrow$,

or, more modestly, of

(ii) extending the collection of states by a single application of the powerset construct, and then treating a state given by a set of states as a family of processes running "conjunctively" in parallel, as in Concurrent Dynamic Logic (Peleg [13]).

The relationship between (i) and (ii) can be explained in the framework of labelled transition systems, on the basis of a transformation $\mathcal{L}$ that is essentially the familiar construction of a deterministic finite automaton from a non-deterministic one (e.g., Hopcroft and Ullman [7]). The transformation $\mathcal{L}$ will play a significant role in our investigations; it is perhaps worth mentioning at this point that it can be employed to extract the DRS's mentioned above from dynamic logic.

1 Background: semantics and information growth

The semantic approach taken in this work is to analyze a constraint as a binary relation on states. Insofar as the notion of a first-order formula is closer to that of a constraint, it is instructive to describe how to translate a formula into a program (that is then interpreted as a binary relation) if only to suggest how such an analysis of constraints can be carried out. For the remainder of the paper, however, we will work directly with programs, rather than first-order formulas, taking for granted the step between constraints and programs (which afterall is subject to considerable variation; see, for example, Fernando [2]). In particular, the present section considers the collection P_0 of programs from quantified dynamic logic.

Let us recall briefly how programs are interpreted in dynamic logic. A signature (= vocabulary) $\mathbf{L}$ is fixed, as is an $\mathbf{L}$-model M and a countable set X of variables. The set $\hat{S}$ of states is then the set of functions $f, g, \ldots$ from X to the universe $|M|$ of M, and programs $p \in P_0$ are interpreted semantically as binary relations $\rho(p) \subseteq \hat{S} \times \hat{S}$ according to

$$\begin{aligned} f\ \rho(x :=?)\ g \quad &\text{iff} \quad f = g \text{ except possibly at } x \\ f\ \rho(A?)\ g \quad &\text{iff} \quad f = g \text{ and } M \models A[f] \\ \rho(p;p') \quad &= \quad \rho(p) \circ \rho(p') \\ \rho(p + p') \quad &= \quad \rho(p) \cup \rho(p') \\ \rho(p^*) \quad &= \quad \text{reflexive-transitive closure of } \rho(p)\ , \end{aligned}$$

where $x \in X$, A is an $\mathbf{L}$-formula with free variables from X, and $\circ$ is relational composition. Rather than extending $\models$ simultaneously to modal $\mathbf{L}$-formulas, it is sufficient to build in tests $([p]\bot)?$ through another primitive operation $\neg$ on which to close P_0, with

$$f\ \rho(\neg p)\ g \quad \text{iff} \quad f = g \text{ and there is no } h \text{ s.t. } f\ \rho(p)\ h\ .$$

Now, an obvious way of formulating the monotonicity of information change is to assert that whenever the binary relation interpreting a program relates an input state s to an output state t, then $s \sqsubseteq t$, for some pre-order $\sqsubseteq$ (marking information growth) on states. As (quantified) dynamic logic supports two kinds of atomic programs (viz., random assignments and tests), information can grow along two different dimensions, that we presently take up in turn. In each case, it will prove useful to modify the set $\hat{S}$ of states described above, passing from the semantic function ρ to a function $[\![\cdot]\!]$ on which for all (modified) states s and t,

$$s \, [\![p]\!] \, t \quad \text{implies} \quad s \sqsubseteq t \, . \tag{1}$$

1.1 Introducing variables with existential force: expansive growth

At any given point in a computation, of which dynamic logic provides an abstraction, only finitely many variables are initialized. Moreover, there is a clear sense that executing a random assignment $x :=?$ at a point in which x is not yet defined "adds" information to the computational state. In fact, in relating dynamic logic to the construction of first-order contexts, it is natural to work with states defined only on finitely many variables (as argued in Fernando [1], appealing, for instance, to Henkin witnesses and back and forth constructions). Accordingly, in place of $\hat{S}$, let us consider the set S_0 of *valuations*

$$S_0 \quad = \quad \{s \mid s \text{ is a function from a finite subset of } X \text{ to } |M|\} \, ,$$

partially ordered by the subfunction (i.e., subset) relation $\subseteq$, and replace the semantic function ρ on P_0 by a function $[\![\cdot]\!]_0$ given in the same inductive fashion as ρ except that $\hat{S}$ is replaced by S_0 and

$$s \, [\![x :=?]\!]_0 \, t \quad \text{iff} \quad x \in \mathrm{dom}(t) \text{ and } s = t \text{ except possibly at } x \, .$$

Observe that monotonicity is (of course) not guaranteed: if an input state s is already defined on x, then $x :=?$ can destroy that binding and spoil (1). But it is simple enough to "guard" all random assignments, by replacing $x :=?$ by the *guarded assignment* $x := *$ that assigns a value to x precisely when initially x is unbound (doing nothing otherwise)

$$x = x? \quad + \quad \neg(x = x?) \; ; \; x :=?$$

(i.e., **if not** $x \downarrow$ **then** $x :=?$). A translation of first-order formulas into programs, all of whose random assignments are guarded, can be constructed by resorting to "marked" and "unmarked" variables (Fernando [2]). In that situation, it is easy to see that (1) holds.

1.2 Imposing constraints on variables: eliminative growth and $\mathcal{L}$

To bring out the information a test $A?$ contributes, it is useful to pass to a more complex notion of a state, embodying an additional dimension of partiality on which tests act. It will turn out to be convenient to be slightly abstract on this point, and to isolate the notion of a so-called *(L-)transition system* — i.e., a triple $\langle S, \{\xrightarrow{l}\}_{l \in L}, s_0 \rangle$ where S is a non-empty set of *states*, $\xrightarrow{l} \subseteq S \times S$ for every $l \in L$, and $s_0 \in S$ is an *initial* state. Observe that the structure $\langle S_0, \{[\![p]\!]_0\}_{p \in P_0}, \emptyset \rangle$ is an instance of a P_0-transition system, the initial state $\emptyset$ being $\subseteq$-minimal. Next, consider the following fairly standard transformation on transition systems (coinciding with the construction of a deterministic finite automation from a non-deterministic one; e.g., Hopcroft and Ullman [7]). Given a transition system $\mathcal{S} = \langle S, \{\xrightarrow{l}\}_{l \in L}, s_0 \rangle$, define for every $l \in L$, a ternary relation $\overset{l}{\Rightarrow}$ on sets a, b of S-states as follows

$$a \overset{l}{\Rightarrow} b \quad \text{iff} \quad b = \{t \in S \mid \exists s \in a \; s \xrightarrow{l} t\} \, .$$

Now form the transition system $\mathcal{L}(\mathcal{S})$ by

(i) defining its set $\mathcal{L}(S)$ of states inductively by

$$\frac{}{\{s_0\} \in \mathcal{L}(S)} \qquad \frac{a \in \mathcal{L}(S) \qquad a \overset{l}{\Rightarrow} b \qquad b \neq \emptyset}{b \in \mathcal{L}(S)} \, ,$$

(ii) interpreting every $l \in L$ by the binary relation $\stackrel{l}{\Rightarrow}$ restricted to $\mathcal{L}(S)$, and

(iii) endowing $\mathcal{L}(\mathcal{S})$ the initial state $\{s_0\}$.

(The reason for restricting the states of $\mathcal{L}(\mathcal{S})$ to this subcollection of Power(S) will become clear later.) If the relevant pre-order for $\mathcal{S}$ is $\sqsubseteq$, then the "natural" pre-order on $\mathcal{L}(\mathcal{S})$ is the so-called *Smyth pre-order* $\sqsubseteq_{Sm}$ given by

$$a \sqsubseteq_{Sm} b \quad \text{iff} \quad (\forall t \in b)\ (\exists s \in a)\ s \sqsubseteq t\ .$$

Note that if $\sqsubseteq$ is $=$, then $\sqsubseteq_{Sm}$ is simply $\supseteq$, marking a growth of information that can be said to be "eliminative" (as opposed to the "expansive" growth described in section 1.1, where $\sqsubseteq$ is $\subseteq$).

Accompanying the abstract notion of a transition system is that of a bisimulation (Park [12]). A relation $R \subseteq S \times S'$ is a *bisimulation* between transition systems $\langle S, \{\stackrel{l}{\rightarrow}\}_{l \in L}, s_0\rangle$ and $\langle S', \{\stackrel{l}{\rightarrow'}\}_{l \in L}, s_0'\rangle$ if it satisfies the "back-and-forth" condition that whenever sRs', then for every $l \in L$,

$$(\forall t \stackrel{l}{\leftarrow} s)\ (\exists t' \stackrel{l}{\leftarrow'} s')\ tRt' \quad \text{and} \quad (\forall t' \stackrel{l}{\leftarrow'} s')\ (\exists t \stackrel{l}{\leftarrow} s)\ tRt'\ .$$

$\langle S, \{\stackrel{l}{\rightarrow}\}_{l \in L}, s_0\rangle$ and $\langle S', \{\stackrel{l}{\rightarrow'}\}_{l \in L}, s_0'\rangle$ are said to be *bisimilar* if there is a bisimulation between them relating the initial states s_0 and s_0'. *Bisimilarity* $\leftrightarroweq$ is the largest bisimulation relating s_0 and s_0', and is guaranteed to exist when $\langle S, \{\stackrel{l}{\rightarrow}\}_{l \in L}, s_0\rangle = \langle S', \{\stackrel{l}{\rightarrow'}\}_{l \in L}, s_0'\rangle$. Although the notion of bisimilarity can (in general) be quite complex, observe that under the translation $\mathcal{L}$, it is not. More precisely,

$$\mathcal{L}\langle S, \{\stackrel{l}{\rightarrow}\}_{l \in L}, s_0\rangle \quad \leftrightarroweq \quad \mathcal{L}\langle S', \{\stackrel{l}{\rightarrow'}\}_{l \in L}, s_0'\rangle$$

iff for all $l \in L$,

$$(\exists s \in S)\ s_0 \stackrel{l}{\rightarrow} s \quad \text{iff} \quad (\exists s' \in S')\ s_0' \stackrel{l}{\rightarrow'} s'\ .$$

Now, returning to the transition system $[\![M]\!]_0 = \langle S_0^M, \{[\![p]\!]_0^M\}_{p \in P_0}, \emptyset\rangle$ constructed in section 1.1 from a first-order model M (fixed in the background, and normally suppressed notationally), the following points spelled out in Fernando [1] are relevant. Whereas over countable models M and N,

$$\begin{aligned} [\![M]\!]_0 \leftrightarroweq [\![N]\!]_0 \quad &\text{iff} \quad M \cong N \\ &\text{iff} \quad [\![M]\!]_0 \cong [\![N]\!]_0\ , \end{aligned}$$

applying $\mathcal{L}$ abstracts away some of the dependence on the first-order model. In particular, if P' is the set of programs in P_0 without an occurrence of Kleene star $\cdot^*$, and if $[\![M]\!]'$ is the P'-transition system obtained from restricting the label set of $[\![M]\!]_0$ to P', then

(i) whether or not M and N are countable,

$$\begin{aligned} \mathcal{L}[\![M]\!]' \leftrightarroweq \mathcal{L}[\![N]\!]' \quad &\text{iff} \quad M \equiv N \ \text{(i.e., } M \text{ and } N \text{ satisfy the same first-order sentences)} \\ &\text{iff} \quad \mathcal{L}[\![M]\!]' \cong \mathcal{L}[\![N]\!]'\ , \end{aligned}$$

and

(ii) as suggested by the last equivalence, $\mathcal{L}[\![M]\!]'$ can (up to $\cong$) be constructed syntactically relative to the first-order theory of M, based on what are essentially the DRS's we met earlier in the introduction.

The logical significance of the transformation $\mathcal{L}$ will be brought out further below.

2 Introducing variables with universal force

The DPL interpretation of implication $p \supset q = \neg(p; \neg q)$ yields the following binary relation on S_0

$$s[\![p \supset q]\!]_0 t \quad \text{iff} \quad t = s \text{ and } (\forall s' \text{ s.t. } s[\![p]\!]_0 s')\ (\exists t')\ s'[\![q]\!]_0 t' \ .$$

As any number of s''s might be accessible from s via $[\![p]\!]_0$, any number of t''s may be involved above. Accordingly, the interpretation above is static — which is to say, the input and output states s and t must be the same. The culprit is the non-determinism of $[\![p]\!]_0$, which suggests a "dual" notion of parallelism, as will come as no surprise to readers familiar with so-called and-or computations (see Peleg [13] and the references cited therein). This dual notion will be introduced, however, not by some conjunction construct, but by an "implication" $\Rightarrow$ between programs. More precisely, let us extend our old collection P_0 of programs by forming a collection P_1 of programs closed under the same constructs as P_0 and, in addition, a new primitive binary construct $\Rightarrow$

$$\frac{p \in P_1 \qquad q \in P_1}{(p \Rightarrow q) \in P_1} \ .$$

2.1 A functional interpretation of implication

Borrowing an idea from Kleene [9] (but then stripping away its "constructive" character), let us witness the $\forall\exists$-clause in $s[\![p \supset q]\!]_0 t$ by a (plain set-theoretic) function f to obtain

$$\begin{aligned} s[\![p \Rightarrow q]\!] t \quad \text{iff} \quad & t = (s, f) \text{ and } f \text{ is a function with domain } \{s' \mid s[\![p]\!]s'\} \\ & \text{s.t. } (\forall s' \in \mathrm{dom}(f))\ s'[\![q]\!] f(s') \ . \end{aligned} \tag{2}$$

Notice that in the right hand side, it is important to store s in t in case there is no s' for which $s[\![p]\!]s'$. On the other hand, if there is an s' for which $s[\![p]\!]s'$, then, further on, we might ignore s when imposing conditions on, for instance, the state(s) u for which $(s, f)[\![A?]\!]u$. (This is particularly plausible if information always increases in the sense of (1), in which case s is subsumed by $f(s')$, for every $s' \in \mathrm{dom}(f)$.) So rather than *adopting (2), let us draw the states from the (inductive) closure* $S_{\rightharpoonup}$ of S_0 under partial functions[3]

$$s \in S_{\rightharpoonup} \quad \text{iff} \quad s \in S_0 \ \text{ or } \ (\exists d \subset S_{\rightharpoonup})\ (\exists c \subset S_{\rightharpoonup}) \ \ s \text{ is a function from } d \text{ to } c \ ,$$

and define for $s, t \in S_{\rightharpoonup}$,

$$\begin{aligned} s[\![p \Rightarrow q]\!] t \quad \text{iff} \quad & (t \text{ is a function with non-empty domain } \{s' \mid s[\![p]\!]s'\} \\ & \qquad \text{and } (\forall s' \in \mathrm{dom}(t)) \ \ s'[\![q]\!] t(s')) \\ & \text{or } (t = s \text{ and there is no } s' \text{ s.t. } s[\![p]\!]s') \ . \end{aligned} \tag{3}$$

The intuition behind $p \Rightarrow q$ then is that every process resulting from p must execute q. To preserve the "conjunctive" character of the processes so spawned, extend the semantics $[\![p]\!]$ of a random assignment or test p by requiring inductively that for $s \in S_{\rightharpoonup} - S_0$ and $t \in S_{\rightharpoonup}$,

$$s[\![p]\!] t \quad \text{iff} \quad \mathrm{dom}(s) = \mathrm{dom}(t) \text{ and } (\forall s' \in \mathrm{dom}(s))\ s(s')[\![p]\!] t(s') \ . \tag{4}$$

The remaining clauses for $s[\![p]\!]t$ can then be asserted uniformly over all $s, t \in S_{\rightharpoonup}$. (That is, ; is still interpreted as $\circ$, $+$ as $\cup$, and $\cdot^*$ as reflexive-transitive closure.) Observe that $\neg p$ amounts semantically to $p \Rightarrow \bot$ (where $[\![\bot]\!] = \emptyset$), thence $p \supset q$ can be derived from $\Rightarrow$, $\bot$ and ;. Also, $\forall x\ p$ can be equated with $(x :=?) \Rightarrow p$.

Proposition 1. *The monotonicity postulated by (1) can be secured for $s, t \in S_{\rightharpoonup}$ and every $p \in P_1$ in which all random assignments are guarded, by defining $\sqsubseteq$ on $S_{\rightharpoonup}$ inductively from $\subseteq$ by*

$$\begin{aligned} s \sqsubseteq t \quad \textit{iff} \quad & (s \textit{ and } t \textit{ are valuations and } s \subseteq t) \textit{ or} \\ & (t \textit{ is a function on states and } (\forall t' \in \mathit{dom}(t))\ s \sqsubseteq t(t')) \textit{ or} \\ & (s \textit{ and } t \textit{ are functions on states with the same domain } d \textit{ and} \\ & \qquad (\forall s' \in d)\ s(s') \sqsubseteq t(s')) \ , \end{aligned}$$

which is evidently transitive.

[3] The class $S_{\rightharpoonup}$ is introduced only for notational convenience, to describe the form of the states needed; it will be replaced by a set S_1 shortly.

2.2 Reducing the higher-order interpretation (enter $\mathcal{L}$ once again)

Working out what the extension above from S_0 to $S_{\rightarrow}$ means brings to mind a remark by G. Kreisel:

> Until the mid fifties, I found this subject [intuitionistic logic] distasteful because ... iterated implications made my head spin. They continue to do so, and the same is true of functions of all finite types.

(Kreisel [10], p. 397). To keep matters from getting out of hand, it is useful to cut down functional states s, observing that "all that really matters" in s is its image, which is treated "conjunctively" in (4). Making these points precise is what this section is all about.

For every $p \in P_1$, let $[\![p]\!]_1 \subseteq S_{\rightarrow} \times S_{\rightarrow}$ be the interpretation of p given in section 2.1, and let S_1 be the set consisting of all objects in $S_{\rightarrow}$ accessible from the empty valuation by interpretations of programs $p \in P_1$

$$S_1 \quad = \quad \{\emptyset\} \cup \{s \mid \exists p \in P_1 \; \emptyset [\![p]\!]_1 s\} \; .$$

Our task is to "reduce" $\langle S_1, [\![\cdot]\!]_1, \emptyset \rangle$ to a more "tractable" transition system $\langle S_2, [\![\cdot]\!]_2, s_0 \rangle$ such that

(A) the "essential structure" of $\langle S_1, [\![\cdot]\!]_1, \emptyset \rangle$ is retained in $\langle S_2, [\![\cdot]\!]_2, s_0 \rangle$

and

(B) the interpretation $[\![\cdot]\!]_2$ can be understood independently of $[\![\cdot]\!]_1$ (i.e., the meaning $[\![p]\!]_2$ of p is intrinsic to S_2).

A notion relevant to (A) is that of a bisimulation, suggesting as a natural candidate for $\langle S_2, [\![\cdot]\!]_2, s_0 \rangle$ the transition system obtained by dividing $\langle S_1, [\![\cdot]\!]_1, \emptyset \rangle$ by the largest bisimulation on $\langle S_1, [\![\cdot]\!]_1, \emptyset \rangle$, call it $\leftrightarrows$. Unfortunately, it is not at all clear that this "reduced" structure is conceptually more tractable. It is true enough that desideratum (A) is met, but as for (B), saying simply that the new states are equivalence classes only worsens matters. What is needed is a helpful characterization of $\leftrightarrows$, but, already over S_0, *distinct* valuations may be related by $\leftrightarrows$, depending on the degree of homogeneity of the underlying first-order model M (Fernando [1]).

Focusing instead on the structural complexity introduced by $\Rightarrow$, consider the "image-collapse" of $[\![\cdot]\!]_1$ to the state set

$$S_2 \quad = \quad S_0 \cup \text{Power}(S_0)$$

induced by the reduction R of functions in S_1 to their (hereditary) images in S_2. That is to say, let R be the least fixed point of

$$\begin{aligned} s_1 R s_2 \quad \text{iff} \quad & (s_1 = s_2 \in S_0) \text{ or} \\ & ((\exists d \subset S_1)\ (\exists c \subset S_1)\ \ s_1 \text{ is a function from } d \text{ onto } c \text{ s.t.} \\ & \qquad s_2 = (c \cap S_0) \cup \bigcup \{t \mid \exists s \in c - S_0 \; sRt\}) \; , \end{aligned}$$

where the first disjunct represents the base case, and the second disjunct represents the inductive (hereditary image) case (justified intuitively by the associativity of conjunction, and broken down similarly into two subcases). Define $[\![\cdot]\!]_2 \subseteq S_2 \times S_2$ by

$$s [\![p]\!]_2 t \quad \text{iff} \quad (\exists s_1, t_1 \in S_1)\ s_1 [\![p]\!]_1 t_1 \text{ and } s_1 R s \text{ and } t_1 R t \; . \tag{5}$$

Is R a bisimulation between $\langle S_1, [\![\cdot]\!]_1, \emptyset \rangle$ and $\langle S_2, [\![\cdot]\!]_2, \emptyset \rangle$? To prove this, it would help if the choice of the representatives s_1 and t_1 in the right hand side of (5) is inessential. Turning to clause (4), it is helpful to think of $s(s')$ and $t(s')$ as processes spawned by s' at successive stages s and t of the construction. Another $s'' \in \text{dom}(s)$ $(= \text{dom}(t))$ can spawn the same process at s — i.e., $s(s') = s(s'')$ — or, for that matter, at t, independently of s, so long as the link to $\text{dom}(s)$ is maintained. But that link is severed by R, in the

aftermath of which, (4) might (following desideratum (B)) be replaced by the condition that for $s \in S_2 - S_0$, $t \in S_2$, and a random assignment or test p,

$$s[\![p]\!]_2 t \quad \text{iff} \quad \exists \text{ function } f \text{ mapping } s \text{ onto } t \text{ s.t. } (\forall s' \in s)\ s'[\![p]\!]_0 f(s') \ . \tag{6}$$

But the problem with (6) is that once s' and s'' beget a (common) child — i.e., once s fails to be 1-1 —, they will be committed to live as one onwards — i.e., t can never be 1-1 (contrary to what is possible in $[\![\cdot]\!]_1$). On the other hand, neither

$$s[\![p]\!]_2 t \quad \text{iff} \quad (\forall s' \in s)\ (\exists t' \in t)\ s'[\![p]\!]_0 t' \tag{7}$$

nor

$$s[\![p]\!]_2 t \quad \text{iff} \quad (\forall s' \in s)\ (\exists t' \in t)\ s'[\![p]\!]_0 t' \ \text{ and } \ (\forall t' \in t)\ (\exists s' \in s)\ s'[\![p]\!]_0 t' \tag{8}$$

will do, as both permit s' multiple offspring, allowing $|t| > |s|$ even if s is the image of a 1-1 function. (Under (6), one slip and "till death do you part", no divorce permitted. Without the true moral grounding of (4), a promiscuous ontology arises from (7) or (8), a Puritan one from (6).)

Simplifying $[\![\cdot]\!]_1$ would seem to be no simple matter. Proceeding from desideratum (B), instead of adopting (5), characterize $[\![\cdot]\!]_2 \subseteq S_2 \times S_2$ independently of $[\![\cdot]\!]_1$ by replacing (3) by

$$\begin{aligned} s[\![p \Rightarrow q]\!]_2 t \quad \text{iff} \quad & (\exists \text{ function } f \text{ with non-empty domain } \{s' \mid s[\![p]\!]_2 s'\} \text{ s.t.} \\ & (\forall s' \in \mathrm{dom}(f))\ \ s'[\![q]\!]_2 f(s') \text{ and} \\ & t = \{t' \in S_0 \mid \exists s'\ f(s') = t'\} \cup \bigcup \{t' \subseteq S_0 \mid \exists s'\ f(s') = t'\}) \\ & \text{or } (t = s \text{ and there is no } s' \text{ s.t. } s[\![p]\!]_2 s') \ , \end{aligned}$$

Furthermore, for a random assignment or test p, replace (4) by any of the pairwise non-equivalent clauses (6), (7) or (8) for $s \in S_2 - S_0$ and $t \in S_2$. (The remaining compound programs are then interpreted as before — i.e., ; by $\circ$, etc.) It is not difficult to show that in the transition system given by (6), S_2 can be simplified to $\mathrm{Power}(S_0)$ by a bisimulation relating the valuation s to $\{s\}$. (Or, going the other direction, under (6), $[\![p]\!]_2$ may be redefined more simply as a subset of $S_0 \times \mathrm{Power}(S_0)$; see the discussion of Peleg [13] in section 2.3.) But the larger question is

(‡) *how are $[\![\cdot]\!]_1$ and the four non-equivalent definitions of $[\![\cdot]\!]_2$ (given by (5), (6), (7) and (8)) related?*

An answer is provided by the transformation $\mathcal{L}$ defined in section 1.2. Observe that $\mathcal{L}(\mathcal{S})$ internalizes the non-determinism of $\mathcal{S}$ into $\mathcal{L}(\mathcal{S})$-states, over which the l-transitions (for every $l \in L$) then (externally) become deterministic (i.e., partial functions). An $\mathcal{L}(\mathcal{S})$-state a is a "disjunctive" set in that a can make a transition so long as some $\mathcal{S}$-state in a can. (Recall that $\mathcal{L}(\mathcal{S})$-states are required to be non-empty.) By contrast, an S_2-state is a "conjunctive" set insofar as, under (5), (6), (7) or (8), every valuation in the set must survive in order for the state to survive. Now, returning to (‡),

Theorem 2. *For each of the four definitions of $[\![\cdot]\!]_2$ given above,*

$$\mathcal{L}\langle S_1, [\![\cdot]\!]_1, \emptyset\rangle \ \leftrightarrows \ \mathcal{L}\langle S_2, [\![\cdot]\!]_2, \emptyset\rangle \ .$$

That is, for all programs $p \in P_1$,

$$(\exists s_1 \in S_1)\ \emptyset[\![p]\!]_1 s_1 \quad \textit{iff} \quad (\exists s_2 \in S_2)\ \emptyset[\![p]\!]_2 s_2 \ . \tag{9}$$

Proof (sketch). The idea, roughly put, is to show that it is sufficient to consider only $\subseteq$-minimal conjunctive sets (— in S_1, functions with $\subseteq$-minimal images —) indexed by $\Rightarrow$, which are the same for all variants of $[\![\cdot]\!]_2$. That is, the two instances of existential quantification in (9) can be restricted to S_0 and higher-order states representing minimal families of processes spawned by p.

Remarks.

(i) The bisimilarity asserted by the theorem can*not* be improved to an isomorphism

$$\mathcal{L}(S_1, [\![\cdot]\!]_1, \emptyset) \;\cong\; \mathcal{L}(S_2, [\![\cdot]\!]_2, \emptyset) \;,$$

as can be seen from considering the programs $x := 0 \;\Rightarrow\; x := 0$ and $x :=? \;\Rightarrow\; x := 0$ (where $x := 0$ is, as usual, $x :=?; x = 0?$). Whether or not an isomorphism holds, modulo $\mathcal{L}$, between specific variants of $[\![\cdot]\!]_2$, we leave as open.

(ii) It is instructive to recall (from section 1.2) that applying the transformation $\mathcal{L}$ on the P_0-transition system $\langle S_0, \{\rho(p)\}_{p \in P_0}, \emptyset\rangle$ underlying quantified dynamic logic leads to an analysis of constraints that is "syntactic" insofar as the DRS's mentioned in the introduction are syntactic. This raises the following problem (a solution to which has so far eluded the present author): give a natural characterization of a DRS for the higher-order extension above. (The fact that bisimilarity in Theorem 2 cannot be strengthened to $\cong$ suggests that the matter can be quite delicate.)

(iii) In view of the abovementioned "conjunctive" and "disjunctive" character of $[\![\cdot]\!]_2$ and $\mathcal{L}$, respectively, the thrust of the theorem can be described as a reduction to a "disjunctive normal form" (licensed by classical, but not intuitionistic, logic). At the heart of the higher-order witnesses introduced in section 2.1 are and/or computations, such as those underlying Concurrent Dynamic Logic (Peleg [13]), to which we turn next.

2.3 Comparison with conjunctive parallelism in Concurrent Dynamic Logic

Without going into the gory details, let us pause to relate the binary construct $\Rightarrow$ to the binary conjunction construct $\cap$ introduced in Peleg [13] to capture a "conjunctive" notion of parallelism, dual to the "disjunctive" non-deterministic construct $+$. The pair $t = (s, f)$ in (2) yields, after replacing f (hereditarily) by its image, an instance of Peleg's *reachability pairs*, sets of which are used to interpret programs. More precisely, the old semantic interpretation function ρ reviewed in section 1 is lifted to a function from (the richer collection of) programs to subsets of $\hat{S} \times \text{Power}(\hat{S})$ according, for instance, to

$$\begin{aligned}
f\ \rho(x :=?)\ U \quad &\text{iff} \quad U = \{g\} \text{ for some } g \in \hat{S} \text{ s.t. } f = g \text{ except possibly at } x \\
f\ \rho(A?)\ U \quad &\text{iff} \quad U = \{f\} \text{ and } M \models A[f] \\
f\ \rho(p;p')\ U \quad &\text{iff} \quad (\exists U')\ f\ \rho(p)\ U' \text{ and} \\
&\qquad \exists \text{ function } F \text{ with domain } U' \text{ s.t.} \\
&\qquad\qquad (\forall g \in U')\ g\ \rho(p')\ F(g) \text{ and} \\
&\qquad\qquad U = \bigcup\{F(g) \mid g \in U'\} \\
f\ \rho(p \cap p \quad &\text{iff} \quad (\exists V, W)\ f\ \rho(p)\ V \text{ and } f\ \rho(p')\ W \text{ and } U = V \cup W \;.
\end{aligned}$$

Setting aside the difference between total functions in $\hat{S}$ and finite functions in S_0, the function ρ can be compared more readily to the variants of $[\![\cdot]\!]_2$ by lifting $\rho(p)$ further to a "left-distributive" binary relation on $\text{Power}(\hat{S})$: for non-empty $s \subseteq \hat{S}$, postulate

$$\begin{aligned}
s\ \rho(p)\ t \quad &\text{iff} \quad \text{there is a function } F \text{ with domain } s \text{ such that} \\
&\qquad (\forall s' \in s)\ s'\ \rho(p)\ F(s') \text{ and } t = \bigcup\{F(s') \mid s' \in s\} \;.
\end{aligned}$$

The clause for $\rho(p;p')$ above then comes closest to the variant of $[\![\cdot]\!]_2$ given by (6). Turning to P_2 and $[\![\cdot]\!]_2$, the construct $\Rightarrow$ yields a notion of conjunction

$$p_{\&} \;=\; (\neg\neg p)\ ;\ (p \Rightarrow \mathit{skip})$$

(where $\neg\neg p$ serves to ensure that some transition through p is possible, and *skip* is a test that always succeeds), from which $\Rightarrow$ can be reconstructed, with the help of $\neg$

$$p \Rightarrow q \;=\; p_{\&}; q \;+\; \neg p \;.$$

The construct $\cdot_{\&}$ differs from $\cap$ not so much in being unary (— one can resort, afterall, to $(\neg\neg p);(\neg\neg q);(p+q)_{\&}$ —), but in being "unbounded" insofar as $p_{\&} \approx p \cap p \cap \cdots$.

3 Discussion: back to constraint programming

As has already been pointed out (in footnote 1), the notions of quantification and parallelism above differ in essential ways from the notions in Saraswat [14]. Extending the system above further with constructs for locality ("hiding") and communication are obvious next steps, although some (common) motivation might be helpful for considering these various notions alongside each other. Otherwise, it may be simplest (and most advisable) to study these notions separately, to avoid confusing the different reasons for which they are of interest. (The program construct $\Rightarrow$ above was introduced, for instance, to overcome a specific linguistic problem raised in Groenendijk and Stokhof [4].)

Viewing the situation from a purely logical point of view, however, let us observe in closing that the extension above is not so much constructive as it is higher-order — departing, as it does, from the constructive spirit of Kleene [9] by imposing neither recursion-theoretic nor proof-theoretic conditions on witnessing $\forall\exists$. Of course, if the underlying first-order model is finite, then all the transition relations would be mechanically computable. Otherwise, the "effective" nature of $\Rightarrow$ (i.e., $\Rightarrow$'s claim to being a *program* construct) becomes problematic, in that over say, the standard first-order model of arithmetic, it gives rise to transition relations that are not r.e. (This is true already for $\neg$, which can be defined as $\cdot \Rightarrow \bot$.) Returning to Saraswat [14], one solution would be to approximate $\Rightarrow$ through interleaving and hiding constructs that keep the transitions r.e. That is, the "true concurrency" in the notion above of a variable with universal force might be reduced to interleaving processes with local variables. A first step in that direction would be to sort out the relationship between dynamic logic and process semantics given by so-called labelled transition systems (for which, see Fernando [3]).

References

[1] Tim Fernando. Transition systems and dynamic semantics. In D. Pearce and G. Wagner, editors, *Logics in AI*, LNCS 633 (subseries LNAI). Springer-Verlag, Berlin, 1992. A slightly corrected version has appeared as CWI Report CS-R9217, June 1992.

[2] Tim Fernando. The donkey strikes back. In *Proc. of the 6th Conference of the European Chapter of the Association for Computational Linguistics*, to appear.

[3] Tim Fernando. Comparative transition system semantics. In E. Börger et al., editors, *Computer Science Logic: Selected Papers from CSL '92*. Springer-Verlag, Berlin, to appear.

[4] J. Groenendijk and M. Stokhof. Dynamic predicate logic. *Linguistics and Philosophy*, 14, 1991.

[5] David Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume 2*. D. Reidel, 1984.

[6] Irene Heim. The semantics of definite and indefinite noun phrases. Dissertation, University of Massachusetts, Amherst, 1982.

[7] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[8] J.A.W. Kamp. A theory of truth and semantic representation. In J. Groenendijk et. al., editors, *Formal Methods in the Study of Language*. Mathematical Centre Tracts 135, Amsterdam, 1981.

[9] S.C. Kleene. On the interpretation of intuitionistic number theory. *J. Symbolic Logic*, 10, 1945.

[10] Georg Kreisel. Proof theory: some personal recollections. In G. Takeuti, *Proof Theory* (second edition). North-Holland, Amsterdam, 1987.

[11] David Nelson. Constructible falsity. *J. Symbolic Logic*, 14, 1949.

[12] David Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. 5th GI Conference*, LNCS 104. Springer-Verlag, Berlin, 1981.

[13] David Peleg. Concurrent dynamic logic. *J. Assoc. Computing Machinery*, 34(2), 1987.

[14] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. Dissertation, Carnegie-Mellon University, 1989, Published by the MIT Press.

A Disjunctive Decomposition Control Schema for Constraint Satisfaction*

Eugene C. Freuder
Paul D. Hubbe

Department of Computer Science
University of New Hampshire
Durham, NH 03824, U.S.A.
ecf@cs.unh.edu
pdh@cs.unh.edu

Abstract

The paper presents a control schema for constraint satisfaction. Several algorithms, old and new, are formulated as instances of this schema by specifying different methods of problem decomposition. This formulation facilitates description and comparison of the algorithms and suggests directions for further research. A new decomposition method is presented that is virtually guaranteed to reduce problem size, while always retaining at least one of the solutions to the original problem.

1 Introduction

A *solution* to a *constraint satisfaction problem* (*CSP*) is an assignment of a value to each problem variable that satisfies all the *constraints*, or restrictions, on which combinations of variables are permitted. We will focus here on binary CSPs where the constraints involve two variables. The potential values for a variable constitute its *domain*. We will assume finite domains.

We propose a disjunctive divide and conquer control schema for constraint satisfaction. The schema encompasses a wide variety of specific algorithms, including a new one presented here. It facilitates presentation and comparative analysis of these algorithms and suggests new algorithmic possibilities. In particular, the problem decomposition offers new opportunities for ordered search in a space of alternative problems, and for parallel and distributed processing. A specific new decomposition technique is presented that is virtually guaranteed to reduce the number of possible solutions, i.e. the number of different ways to assign a value to each variable, while always retaining at least one of the actual solutions to the original problem.

Our basic CSP algorithm schema can be stated very simply:

Decomposition Algorithm Schema:
Place the initial problem on the Agenda
Until Agenda empty:
    Remove a problem P from Agenda
    If P has only instantiated variables
        then Exit with their values
        else
            Decompose P into a set of subproblems $\{P_i\}$
            Place each non-empty P_i onto the Agenda
Exit with no solution

*This material is based upon work supported by the National Science Foundation under Grant No. IRI-9207633.

Initially all variables are *uninstantiated*; the decomposition methods mark variables as *instantiated*. Intuitively, the instantiated variables are the variables for which we have chosen values. Upon exit the cross product of the instantiated variable domains is the set of reported solutions. (This is not necessarily all the solutions, but some of our algorithms will naturally find sets of solutions even while searching for a first solution.) A problem is *empty* if any of its domains is empty.

We impose the following three conditions on $\{P_i\}$:

1. *Soundness*: Any solution to any P_i is a solution to P.
2. *Termination*: Each of the $\{P_i\}$ is smaller than P. (*Problem size* can be measured as the product of the domain sizes for each variable, i.e. the number of combinations of values that could be generated as potential solutions.)
3. *Semi-completeness*: If there is a solution to P, then there will be a solution to at least one of the $\{P_i\}$.

This *disjunctive decomposition* breaks a problem into subproblems in a manner that guarantees that the decomposition algorithm schema will produce a solution to solvable problems and terminate without a solution when none exists. Notice that if we are only looking for one solution, we do *not* need to require that every solution to P will be a solution to some P_i. (*Conjunctive decompositions* break a problem into subproblems such that all the subproblems must be solved, and the solutions must fit together properly, for the original problem to be solved.)

This schema immediately suggests two avenues of exploration:

1. How is the decomposition performed?
2. How is the agenda organized?

Different answers to these questions produce a family of divergent algorithms.

Notice that if the agenda is maintained as a stack we have a form of depth-first search, which does not need to present a serious space problem. Stack size requirements are O(nD), for n variables and a maximum of D problems in any decomposition $\{P_i\}$. (In fact if we represent all but the first component as a continuation, generating the individual subproblems as needed, stack size can be reduced to O(n).) On the other hand, if we are more flexible in the agenda ordering, it permits opportunities for heuristic ordering that may save processing time. (If ordering is limited to ordering a set $\{P_i\}$ before placing it on the agenda, stack size requirements can still be O(nD).)

All of our examples will assume a stack organization for the agenda. However, that still leaves open questions about which problems to place next on the stack, and in what order.

We will indicate how versions of five specific algorithms can be formulated using this schema:

1. backtracking (BT) [Golumb and Baumert, 1965]: We use this basic algorithm to introduce the schema.

2. forward checking (FC) [Haralick and Elliott, 1980]: This is one of the most successful CSP techniques. The effective minimal domain size variable ordering [Haralick and Elliott, 1980] is naturally incorporated as a decomposition decision.

3. network consistency (NC) [Mackworth, 1977]: This algorithm explicitly operated in a recursive divide and conquer form, alternating local consistency processing with variable domain splitting. The schema formulation helps suggest a variety of NC variations to explore, and clarifies the relationship between NC and FC.

4. backtracking with cross product representation (BT-CPR) [Hubbe and Freuder, 1992]: The schema formulation, similar to that used in the original presentation of the algorithm, facilitates viewing backtracking as a degenerate case of backtracking with CPR. This in turn facilitates the demonstration that adding CPR can not increase the number of *constraint checks* (a standard measure of CSP algorithm performance) when searching for all solutions (and may reduce the number of checks significantly). A similar formulation is possible for forward checking with CPR.

5. inferred disjunctive constraints (IDC) [Freuder and Hubbe, to apear]: The new IDC algorithm was explicitly intended as a decomposition algorithm. It takes advantage of the fact that some solutions may be thrown away in the search for a single solution. The comparison with forward checking that the schema facilitates provides insight into the effective use of this technique. The technique can be shown to virtually always reduce problem size, i.e. the sum of the sizes of the subproblems will virtually always be less than the size of the decomposed problem.

Decomposition methods can be combined. IDC incorporates aspects of FC. An algorithm that uses heuristics to alternatively choose between FC and IDC decomposition during search has proven superior to either IDC or FC alone (testing all three with minimal domain size variable ordering) on some very hard problems [Freuder and Hubbe, to appear].

In the following five sections we specify the decomposition techniques that effectively define each of the five basic algorithms listed above. Plugging each decomposition technique into the decomposition schema produces a version of one of the algorithms. In each case we specify the decomposition abstractly and then illustrate it with a simple example. Section 7 discusses some theoretical aspects of the efficacy and efficiency of these algorithms. The final section proposes some directions for further work.

The abstract decomposition descriptions will refer to a *decomposed problem*, P, with n variables. Each of the subproblems in the decomposition will be specified by describing how to construct them from P. The subproblems are to placed on the decomposition schema's stack so that they reside on the stack in the same order as they are specified. We will often refer to the "first" uninstantiated variable, or the "first" value in a domain, assuming the variables and values are stored in some order. We specify "first" rather than "any" in order to present a more specific algorithm, rather than another schema, parameterized around the method of variable and value choice. However, we could also impose a heuristic variable or value search ordering scheme to choose the variables or values (or provide their initial ordering), and such schemes are of considerable interest.

We will use as an example a simple coloring problem. The variables are countries, the values are colors, the constraints specify that neighboring countries cannot have the same color. We will have four countries (variables): W, X, Y and Z. Each country has three possible colors (values): r(ed), b(lue), and g(reen). The countries are arranged in a "ring": W neighbors X, X neighbors Y, Y neighbors Z and Z neighbors W.

Coloring problems and subproblems will be represented by listing the domains for W, X, Y and Z in order. Thus the original problem can be represented:

r b g
r b g
r b g
r b g

The domains of instantiated variables will be represented in italics. We will carry out the decomposition depth-first until a solution is found. We will show all the non-empty sibling subproblems for each decomposition, even though in practice we need not generate all siblings at once. Empty subproblems will not be shown.

This example is purposely trivial for pedagogical purposes. It is *not* intended to illustrate the relative merits of the different algorithms. However, it may provide some insight into their potential, as well as their operation.

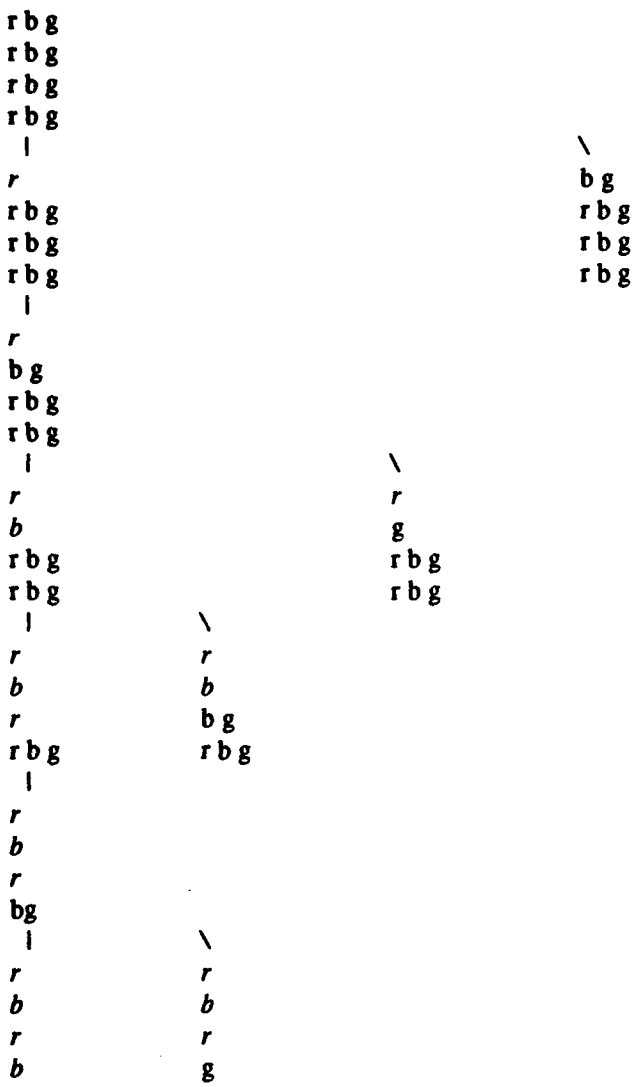
2 Backtracking

Decomposition:

1. The *instantiated subproblem.*
Mark the first uninstantiated variable, V, instantiated. Make its domain the first value, v, in the domain of V. If v is inconsistent with any of the (single) values in the domains of any of the previously instantiated variables, also remove v from the domain of V, leaving the instantiated problem empty (to be discarded by the algorithm schema).

2. The *remainder subproblem.*
Remove the value v from the domain of V.

Example: (Remember that we are not showing empty subproblems.)



3 Forward Checking

Decomposition:

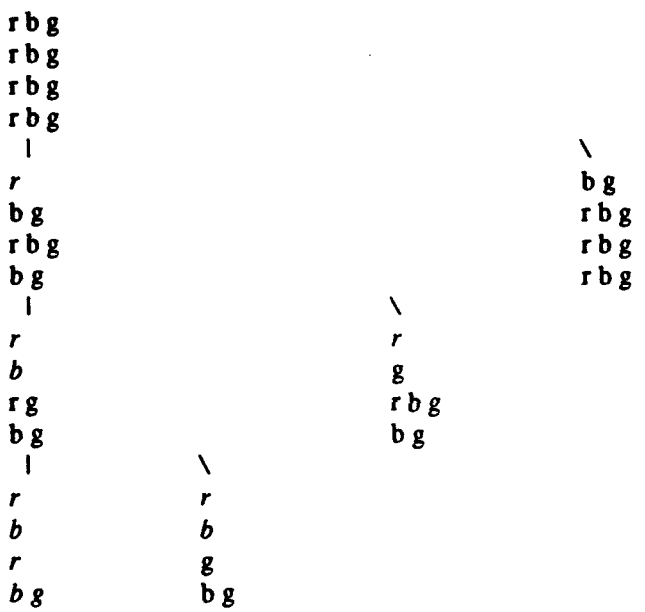
1. The *precluded subproblem* .

Mark the first uninstantiated variable, V, instantiated. Make its domain the first value, v, in the domain of V. Remove values inconsistent with v from the domains of the remaining uninstantiated variables. As an obvious non-standard refinement, if there is only one uninstantiated variable, it too can be marked instantiated.

2. The *remainder subproblem.*

Remove the value v from the domain of V. (Same as in the backtracking decomposition.)

Example:



4 Network Consistency

Decomposition:

1. The *first divided subproblem* .

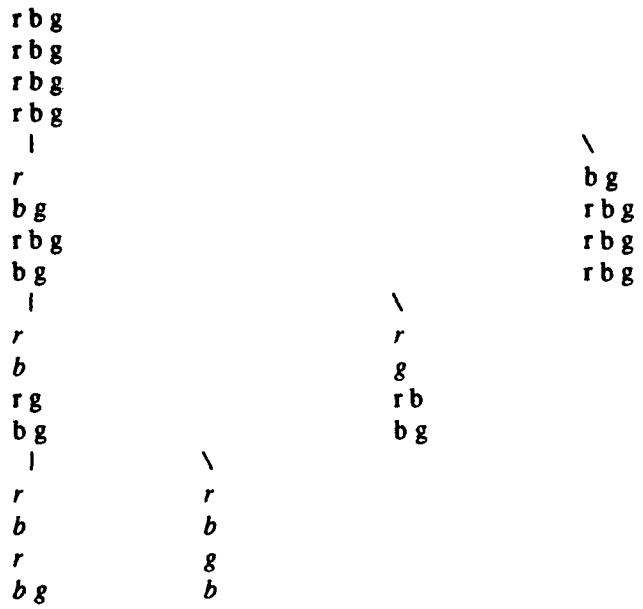
Remove half the values from the domain of the first uninstantiated variable, V. Subject the subproblem to arc consistency processing, which may further reduce variable domains. Finally, mark any variables with single value domains instantiated, and if there is only one uninstantiated variable, mark it instantiated also.

2. The *second divided subproblem.*

Remove from the domain of V the values in the domain of V in the first divided subproblem. As with the first divided subproblem: Subject the subproblem to arc consistency processing, which may further reduce variable domains. Finally. mark any variables with single value domains instantiated, and if there if only one uninstantiated variable, mark it instantiated also.

(Note that in employing this decomposition in the decomposition algorithm schema the arc consistency processing does not actually have to be done until we take the subproblem off the agenda.)

Example:



5 Backtracking With Cross Product Representation

Decomposition:

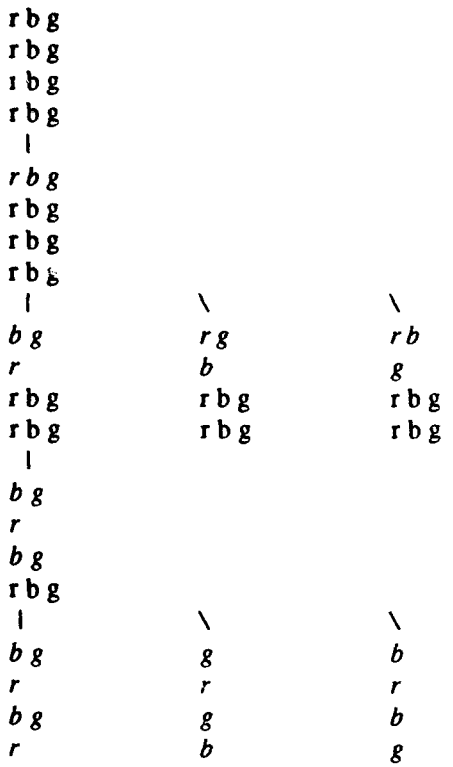
The *CPR subproblems.*

a. Split: For each value v in the first uninstantiated variable, V, create a subproblem where the domain of V is restricted to v, and the domains of each instantiated variable are restricted to those values consistent with v.

b. Merge: If any set of subproblems differ only in the domain of V, merge them into a single subproblem where the domain of V is the union of all their individual V domains (and the other domains are the same as they are in each of these subproblems).

Mark V instantiated in each subproblem.

Example:



6 Inferred Disjunctive Constraint

Decomposition:

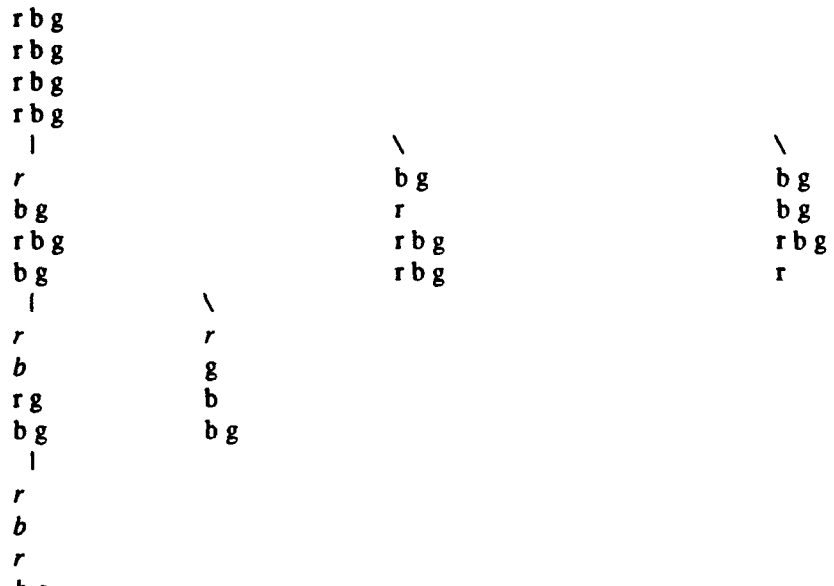
1. The *precluded subproblem.*

Mark the first uninstantiated variable, V_1, instantiated. Make its domain the first value, v, in the domain of V_1. Remove values inconsistent with v from the domains of the remaining uninstantiated variables. If there is only one uninstantiated variable, it too can be marked instantiated. (Same as in the forward checking decomposition.)

2. The *excised subproblems.*

For each remaining uninstantiated variable V_i, i = 2 to n, create a subproblem by removing v from the domain of V_1, removing any values inconsistent with v from the domains of V_2 through V_{i-1} and removing any values consistent with v from the domain of V_i. (Note any variables that do not share a constraint with V_1 will automatically lead to empty subproblems.)

Example:



7 Theory

All of these decompositions meet the three conditions laid out in the first section, and thus when the decomposition algorithm schema uses them it will terminate and find a solution to solvable problems. Only the IDC decomposition takes advantage of the fact that some solutions can be thrown away as long as not all are thrown away. The others do not throw away any solutions, and thus if the algorithms employing these decompositions continue to explore the search space after finding solutions ("pretending" to fail) they will find all solutions.

CPR, a relatively new decomposition, is based on the insight that sets of incomplete solutions may be represented and efficiently processed in a cross product representation. It is proven analytically in [Hubbe and Freuder, 1992] that BT-CPR never requires more constraint checks than BT, when searching for all solutions, or proving that none exist. A similar result is obtained for CPR in conjunction with FC.

The new decomposition, IDC, is based on the hypothesis that, if P is solvable, either there will be a solution involving v for V, or there will be a solution involving a value inconsistent with v. It is proven in [Freuder and Hubbe, to appear] that:

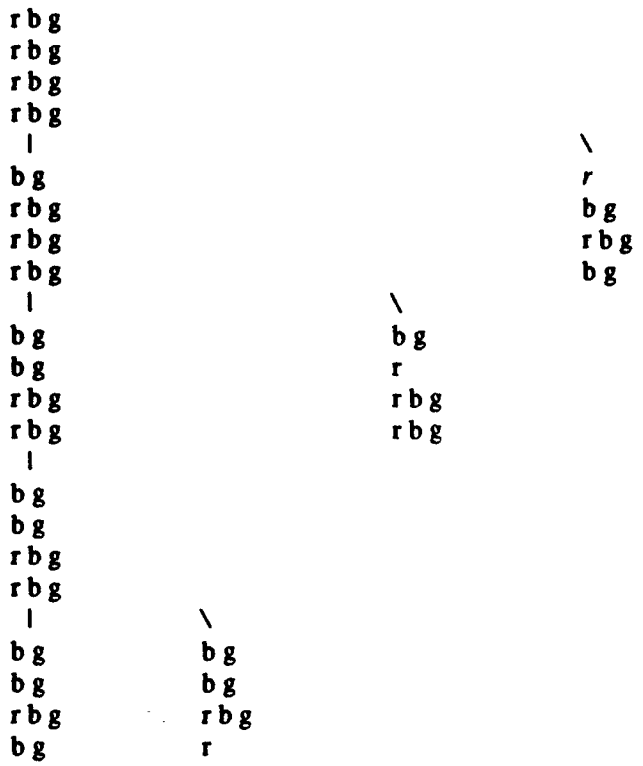
1. IDC will find a solution if one exists. It may throw away some solutions, but often we are only looking for one anyway.
2. The size of the IDC decomposition (the sum of the sizes of the subproblems) will always be smaller than the size of the decomposed problem (except in the degenerate case when the instantiated value is the only value in its domain, and all other values are consistent with it). None of the other decomposition schemes can make such a strong reduction claim.
3. The size of the IDC decomposition will always be less than the size of the FC decomposition by an amount equal to the size of the *consistent subproblem*. If v is the value in the domain of the first variable, V, in the precluded subproblem, the consistent subproblem is formed from the decomposed problem by removing v from the domain of V and removing from all other domains any value inconsistent with v.

This theoretical analysis of IDC is facilitated by another decomposition process. We will describe this process and then present an example, again using our coloring problem. (The forward checking decomposition in the example is shown in the reverse order to that shown in Section 3.) Compare the leaves of the tree in the example with the first decomposition in Section 6.

Description:

First carry out a forward checking decomposition. Call the variable instantiated in the precluded subproblem V and the value in its domain v. Now decompose the remainder problem into two subproblems using a variation of the NC decomposition: divide the domain of the variable after V, not in half, but into two pieces, one containing all values inconsistent with v, the other containing all the values consistent with v. Repeat this NC-like decomposition process on the second subproblem, the one containing the consistent values, dividing the domain of the next variable into two pieces. Continue in this manner until all domains have been so divided. The leaves of the resulting decomposition tree will be the subproblems of the IDC decomposition plus the consistent subproblem. Now observe that given any solution to the consistent subproblem we can substitute v for the value of the first variable and still have a solution.

Example:



8 Further Work

Viewing these algorithms as instances of the decomposition schema helps us to compare them, and suggests new variations. For example, if we alter NC to subdivide a domain of d values into d pieces instead of 2 pieces, and reduce the amount of arc consistency processing appropriately, we arrive at forward checking. Is there another way to subdivide the domain that outperforms both algorithms for an interesting class of problems?

Ordering heuristics for subproblem consideration provide a new avenue to explore. We can view ourselves as searching in a metalevel "subproblem space". This subproblem space obviously lends itself to distributed and parallel processing, especially given the disjunctive nature of our decompositions. Sophisticated constraint languages may someday mix and match decomposition techniques as most appropriate for the problem at hand. Most intriguing of all is the possibility that useful new forms of decomposition are waiting to be discovered.

References

[Freuder and Hubbe, to appear] E. Freuder and P. Hubbe. Using inferred disjunctive constraints to decompose constraint satisfaction problems. *Proceedings of the Thirteenth IJCAI.*

[Golumb and Baumert, 1965] S. Golumb and L. Baumert. Backtrack programming. *JACM 12.* 516-524.

[Haralick and Elliott, 1980] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14.* 263-313.

[Hubbe and Freuder, 1992] P. Hubbe and E. Freuder. An efficient cross product representation of the constraint satisfaction problem search space. *Proceedings of the Tenth National Conference on Artificial Intelligence.* 421-427.

[Mackworth, 1977] A. Mackworth. On reading sketch maps. *Proceedings of the Fifth IJCAI.* 598-606.

Terminological Reasoning with Constraint Handling Rules

Thom Frühwirth*
ECRC, Arabellastrasse 17
D-W-8000 Munich 81, Germany
thom@ecrc.de

Philipp Hanschke†
DFKI, Postfach 2080
D-W-6750 Kaiserslautern, Germany
hanschke@dfki.uni-kl.de

Abstract

Constraint handling rules (CH rules) are a flexible means to implement 'user-defined' constraints on top of existing host languages (like Prolog and Lisp). Recently, M. Schmidt-Schauß and G. Smolka proposed a new methodology for constructing sound and complete inference algorithms for terminological knowledge representation formalisms in the tradition of KL-ONE. We propose CH rules as a flexible implementation language for the consistency test of assertions, which is the basis for all terminological reasoning services.

The implementation results in a natural combination of three layers: (*i*) a constraint layer that reasons in well-understood domains such as rationals or finite domains, (*ii*) a terminological layer providing a tailored, validated vocabulary on which (*iii*) the application layer can rely. The flexibility of the approach will be illustrated by extending the formalism, its implementation and an application example (solving configuration problems) with attributes, a new quantifier and concrete domains.

1 Introduction

Constraint logic programming (CLP) [JaLa87, Sar89, HS90, Coh90, VH91] combines the advantages of logic programming and constraint solving. In logic programming, problems are stated in a declarative way using rules to define relations (predicates). Problems are solved by the built-in *logic programming engine* (LPE) using backtrack search. In constraint solving, efficient special-purpose algorithms are used to solve problems involving distinguished relations referred to as constraints. Constraint solving is usually 'hard-wired' in a built-in *constraint solver* (CS). While efficient, this approach makes it hard to extend or specialize a given CS, combine it with other CS's or build a CS over a new domain.

Constraint handling rules (CH rules) [Fru92] are a language *extension* providing the user (application-programmer) with a declarative and flexible means to introduce *user-defined* constraints (in addition to *built-in* constraints of the underlying host language). In this paper the host language is Prolog, a CLP language with equality over Herbrand terms as built-in constraint. CH rules define *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence (e.g. `X>Y,Y>X <=> false`). Propagation adds new constraints which are logically redundant but may cause further simplification (e.g. `X>Y,Y>Z ==> X>Z`). When repeatedly applied by a constraint handling engine (CHE) the constraints may become solved as in a CS (e.g. `A>B,B>C,C>A` results in `false`).

CHIP was the first CLP language to introduce some constructs (demons, forward rules, conditionals) [D*88] for user-defined constraint *handling* (solving, simplification, propagation). These various constructs have been generalized into CH rules. CH rules are based on guarded rules, as can be found in concurrent logic programming languages [Sha89], in the Swedish branch of the Andorra family [HaJa90], Saraswats cc-framework of concurrent constraint programming [Sar89], and - with similar motivation as ours - in the 'Guarded Rules' of [Smo91]. However all these languages (except CHIP) lack features essential to define non-trivial constraint handling, namely handling conjunctions of constraints and defining constraint propagation. CH rules provide these two features by multiple heads and propagation rules.

*Supported by ESPRIT Project 5291 CHIC
†Supported by BMFT Research Project ARC-TEC (Grant ITW 8902 C4)

Terminological formalisms based on KL-ONE [BS85] are used to represent the terminological knowledge of a particular problem domain on an abstract logical level. To describe this kind of knowledge, one starts with atomic concepts and roles, and defines new concepts using the operations provided by the language.

simple-device isa device and some connector is interface.

These intensionally defined concepts can be considered as unary predicates, and roles as binary predicates over individuals. The limited expressiveness of terminological formalisms enables decision procedures for a number of interesting reasoning problems like consistency of assertions and classification of concepts.

The key idea of [ScSm91] for constructing such inference algorithms is to reduce all inference services to a consistency test which can be regarded as a tuned tableaux calculus. We propose CH rules as a flexible implementation layer for this consistency test. These CH rules directly reflect the rules of the tableaux calculus.

In [BaHa91, Han92] we have shown how a terminological formalism can be parametrized by a *concrete domain*, e.g. constraints over rational numbers. This and other extensions carry over to the implementation with CH rules in a straight-forward manner. Concrete domains can be either also implemented by CH rules or provided as built-in constraints of the host language. In this way we obtain a fairly natural combination of three knowledge representation layers on a common implementational basis.

2 Constraint Logic Programming with Constraint Handling Rules

2.1 Syntax

A CLP+CH program is a finite set of clauses from the CLP language and from the language of CH rules. Atoms and terms are defined as usual. There are two classes of distinguished atoms, built-in constraints and user-defined constraints.

A *CLP clause* is of the form

H :- $B_1, \ldots B_n$. $(n \geq 0)$

where the head H is an atom but not a built-in constraint, the body $B_1, \ldots B_n$ is a conjunction of atoms called *goals*. There are two kinds of CH rules (call declarations [Fru92] are not described in this paper).

A *simplification* CH rule is of the form

$H_1, \ldots H_i$ <=> $G_1, \ldots G_j \mid B_1, \ldots B_k$.

A *propagation* CH rule is of the form

$H_1, \ldots H_i$ ==> $G_1, \ldots G_j \mid B_1, \ldots B_k$. $(i > 0, j \geq 0, k \geq 0)$

where the multi-head $H_1, \ldots H_i$ is a conjunction of user-defined constraints and the guard $G_1, \ldots G_j$ is a conjunction of atoms which neither are, nor depend on, user-defined constraints.

2.2 Semantics

Declaratively, CLP programs are interpreted as formulas in first order logic. A CLP+CH program P is a conjunction of universally quantified clauses. A CLP clause is an implication

$H \rightarrow B_1 \wedge \ldots B_n$.

A simplification CH rule is a logical equivalence provided the guard is true

$(G_1 \wedge \ldots G_j) \rightarrow (H_1 \wedge \ldots H_i \leftrightarrow B_1 \wedge \ldots B_k)$. A propagation CH rule is an implication provided the guard is true

$(G_1 \wedge \ldots G_j) \rightarrow (H_1 \wedge \ldots H_i \rightarrow B_1 \wedge \ldots B_k)$.

Extending a CLP language with CH rules preserves its declarative semantics, as *correct* CH rules are logically redundant with regard to the CLP program. CH rules are not supposed to change the meaning of a program, but the way it is executed.

The operational semantics of CLP+CH can be described by a transition system. In the following we do not distinguish between sets and conjunctions of atoms. A *constraint store* represents a set of constraints. Let C_U and C_B be two constraint stores for user-defined and built-in constraints respectively. Let Gs be a set of goals. A *computation state* is a tuple

$< Gs, C_U, C_B >$.

The *initial state* consists of a query Gs and empty constraint stores,

$< Gs, \{\}, \{\} >$.

A *final state* is either *successful* (no goals left to solve),

$< \{\}, C_U, C_B >$,

or *failed* (due to an inconsistent constraint store),

$< Gs, \texttt{false}, C_B >$ or $< Gs, C_U, \texttt{false} >$.

The union of the constraint stores in a successful final state is called *conditional answer* for the query Gs, written *answer*(Gs).

The built-in CS works on built-in constraints in C_B and Gs, the user-defined CS on user-defined constraints in C_U and Gs using CH rules and the LPE on goals in Gs and C_U using CLP clauses.

The following *computation steps* are possible to get from one computation state to the next.

The built-in CS updates the constraint store C_B if a new constraint C was found in Gs. To *update* the constraint store means to produce a new constraint store C'_B that is logically equivalent to the conjunction of the new constraint and the old constraint store.

Solve $< \{C\} \cup Gs, C_U, C_B > \longmapsto < Gs, C_U, C'_B >$
if $(C \wedge C_B) \mapsto C'_B$

The CHE simplifies and propagates from user-defined constraints in Gs and C_U if a new user-defined constraint was found or the built-in constraint store had been updated so that a guard can be satisfied. To *simplify* user-defined constraints $(H' \cup H'')$ means to replace them by B if $(H' \cup H'')$ matches the head H of a simplification CH rule H <=> G | B and G is satisfied. To *propagate from* user-defined constraints $(H' \cup H'')$ means to add B to Gs if $(H' \cup H'')$ match the head H of a propagation CH rule H ==> G | B and G is satisfied. A guard G is *satisfied* if its local execution does not involve user-defined constraints and the result *answer*(G) is entailed (implied) by the built-in constraint store C_B.

Simplify $< H' \cup Gs, H'' \cup C_U, C_B > \longmapsto < Gs \cup B, C_U, C_B >$
if $(H$ <=> $G \mid B) \in P$ *and* $C_B \rightarrow H = (H' \cup H'') \wedge answer(G)$

Propagate $< H' \cup Gs, H'' \cup C_U, C_B > \longmapsto < Gs \cup B, H' \cup H'' \cup C_U, C_B >$
if $(H$ ==> $G \mid B) \in P$ *and* $C_B \rightarrow H = (H' \cup H'') \wedge answer(G)$

The LPE unfolds goals in Gs. To *unfold* a goal H' means to look for a clause $H :- B$ and to replace the H' by $(H = H')$ and B. As there are usually several clauses for a goal, unfolding is nondeterministic and thus a goal can be solved in different ways using different clauses.

Nondeterministic Unfold $< \{H'\} \cup Gs, C_U, C_B > \longmapsto < Gs \cup B, C_U, \{H = H'\} \cup C_B >$
if $(H$:- $B) \in P$

CHEER, an interpreter for CH rules is available based on ECRC's Eclipse Prolog utilizing its delay-mechanism and built-in meta-predicates to create, inspect and manipulate delayed goals. In such a sequential implementation, the transitions are tried in the above textual order. We wrote real-life constraint handlers for booleans, finite domains (à la CHIP [D*88]), temporal reasoning (quantitative and qualitative constraints over time points and intervals [Fru93]) and real closed fields (à la CLP(R) [J*92]). Typically it took only a few days to produce a prototype, since one can directly express how constraints simplify and propagate without worrying about implementation details. If inefficient, once the handler has been tested and 'tuned' as required, it can be safely reworked in a low-level language.

3 Terminological Reasoning

In this section we will recall the concept language $\mathcal{ALC}$ [ScSm91] as our basic terminological logic (TL) and show its implementation in CH rules. Section 4 will then proceed with some useful extensions of this formalism demonstrating the flexibility of the CH rules approach.

3.1 Terminology

A terminology (T-box) consists of a finite, cycle free set of *concept definitions* "C **isa** s" where C is the newly introduced concept name and s is a concept term constructed from concept names and roles. Inductively, *concept terms* are defined as follows:

1. Every concept name C is a concept term.

2. If s and t are concept terms and R is a role name then the following expressions are concept terms:
   s **and** t (conjunction), s **or** t (disjunction), **nota** s (complement).
   every R **is** s (value restriction), **some** R **is** s (exists-in restriction)

An interpretation $\mathcal{I}$ with a set $\text{dom}_\mathcal{I}$ as domain interprets a concept name C as a set $C^\mathcal{I} \subseteq \text{dom}_\mathcal{I}$ and a role name R as a set $R^\mathcal{I} \subseteq \text{dom}_\mathcal{I} \times \text{dom}_\mathcal{I}$. It can be lifted to concept terms in a straight-forward manner: conjunction, disjunction, and complement are interpreted as set intersection, set union, and set complement wrt $\text{dom}_\mathcal{I}$, respectively, and

$a \in (\textbf{every } R \textbf{ is } s)^\mathcal{I}$ iff, for all $b \in \text{dom}_\mathcal{I}$, $(a,b) \in R^\mathcal{I}$ implies $b \in s^\mathcal{I}$, and

$a \in (\textbf{some } R \textbf{ is } s)^\mathcal{I}$ iff, there is some $b \in \text{dom}_\mathcal{I}$ such that $(a,b) \in R^\mathcal{I}$, $b \in s^\mathcal{I}$.

An interpretation is a *model* of a terminology T if $C^\mathcal{I} = s^\mathcal{I}$ for all "C **isa** s" $\in T$.

Example: The domain of a configuration application comprises at least devices, interfaces, and configurations. The following concept definitions express that these are disjoint sets.

```
primitive device.[1]
interface isa nota device.
configuration isa nota (interface or device).
```

Let's assume that a simple device has at least one interface. So we introduce a role and employ the exists-in restriction.

```
role connector.
simple_device isa device and some connector is interface.
```

□

3.2 Assertions and Reasoning Services

Objects are (Herbrand) constants or variables. Let a, b be objects, R a role, and C a concept term. Then $b : C$ is a *membership assertion* and $(a,b) : R$ is a *role-filler assertion*. An *A-box* is a collection of membership and role-filler assertions.

Example (contd): So we can introduce instances of devices and interfaces.

```
dev2:device, inter1:interface, (dev1,inter1):connector.
```

□

An *interpretation of an A-box* $\mathcal{A}$ is a model of the underlying terminology that, in addition, maps herbrand constants to elements of $\text{dom}_\mathcal{I}$. For these constants we adopt the unique name assumption. An A-box $\mathcal{A}$ is *consistent* if there is an interpretation $\mathcal{I}$ and a variable assignment σ : *objects* $\rightarrow \text{dom}_\mathcal{I}$ such that all assertions of $\mathcal{A}$ are satisfied, i.e., $(a\sigma^\mathcal{I}, b\sigma^\mathcal{I}) \in R^\mathcal{I}$ and $b\sigma^\mathcal{I} \in C^\mathcal{I}$, for all $(a,b) : R$ and $b : C$ in $\mathcal{A}$. An object a is a *member of a concept* C iff for all models $\mathcal{I}$ of the terminology all assignments σ : *objects* $\rightarrow \text{dom}_\mathcal{I}$ that satisfy $\mathcal{A}$ also satisfy $a : C$. A concept C_1 *subsumes* a concept C_2 iff for all models $\mathcal{I}$ of the terminology $C_1^\mathcal{I} \supseteq C_2^\mathcal{I}$. Figure 1 shows the subsumption graph of the terminology developed in Section 3 and 4.

The (in)consistency test is the central reasoning service for terminological knowledge representation systems with complete inference algorithms. Various other services can be reduced to this test [Hol90]. In particular, the subsumption (and similarly membership) services can be implemented on the basis of the consistency test of A-boxes:

1. A concept C_1 subsumes a concept C_2 iff an A-box consisting just of the membership assertion a : C_2 **and nota** C_1 is inconsistent.

2. An object a is a member of C wrt the A-box $\mathcal{A}$ iff $\{a : \textbf{nota } C\} \cup \mathcal{A}$ is inconsistent.

[1] This declaration introduces **device** as a primitive concept name that is not defined any further.

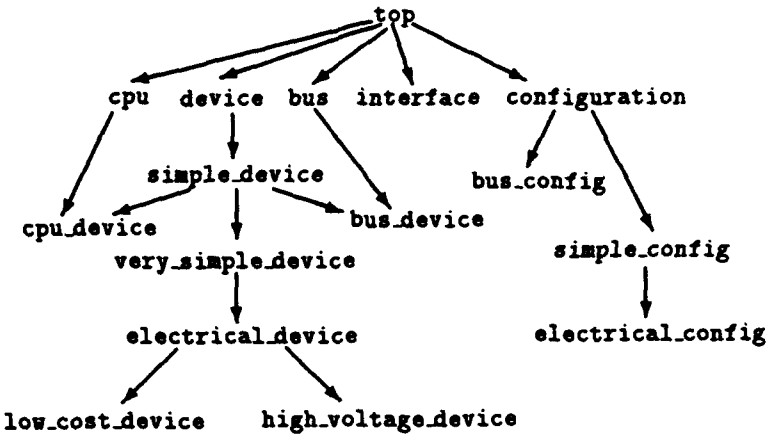


Figure 1: Subsumption Graph of the Example Terminology

3.3 CLP+CH(TL)

Roughly, the consistency test of A-boxes works as follows.

1. Simplify and propagate the assertions in the A-box to make the knowledge more explicit.
2. Look for obvious contradictions (clashes) such as "`a:B, a:nota B`".

Both steps can be directly mapped to CH rules by regarding assertions as user-defined constraints:

1. `I:nota (S or T) <=> I:(nota S and nota T).`
   `I:nota (S and T) <=> I:(nota S or nota T).`
   `I:nota nota S <=> I:S.`
   `I:nota every R is S <=> I:some R is nota S.`
   `I:nota some R is S <=> I:every R is nota S.`
   These replacement rules show how the complement operator can be pushed towards the leaves of a concept term.

   The conjunction rule generates two new, smaller assertions: `I:S and T <=> I:S,I:T.`
   Disjunction is treated by two CLP clauses: `I:S or T :- I:S.`
   `I:S or T :- I:T.`
   An exists-in restriction generates a new object: `I:some R is S <=> (I,J):R, J:S.`
   A value restriction has to be propagated to all role fillers: `I:every R is S, (I,J):R ==> J:S.`
   Note that for termination it is essential that this propagation rule is applied only once per matching pair of membership and role-filler assertions.

   The unfolding rules expand concept names to their definitions:
   `I:C <=> (C isa S) | I:S.`
   `I:nota C <=> (C isa S) | I:nota S.`
2. `I:nota S,I:S ==> false.` For $\mathcal{ALC}$ we need only this single clash rule.

4 Extensions

In a number of papers the above idea of a tableaux based consistency test as the central reasoning service has been successfully applied to terminological logics with various other language constructs (e.g., [HNS90, Hol90]). This flexibility carries over to extensions of our implementation.

4.1 Functional Roles

Roles are interpreted as an arbitrary binary relation over $\mathrm{dom}_\mathcal{I}$. *Attributes* (also called features) are functional roles, i.e., their interpretation is the graph of a partial function $\mathrm{dom}_\mathcal{I} \rightarrow \mathrm{dom}_\mathcal{I}$. Assuming declarations of attributes of the form **attribute** F, F an attribute name, we just have to extend our implementation by

```
(I, J1):F, (I, J2):F  ==>  attribute F | J1=J2.
```

Example (contd): Now we are ready to define a simple configuration which consists of two distinguished devices:

```
attribute component_1.
attribute component_2.
simple_config isa configuration and
          some component_1 is simple_device and
          some component_2 is simple_device.
```

Extending the above A-box by

```
config1:simple_config, (config1,dev1):component_1, and (config1,dev2):component_2
```

the membership service can derive that **dev1** and **dev2** are simple devices. □

A more local way to specify functionality of roles is provided through concept terms of the form "**at most one** R", R a role name.[2] An $a \in \mathrm{dom}_\mathcal{I}$ is an element of (**at most one** R)$^\mathcal{I}$ if there is at most one R-role filler for a. This is implemented through

```
I:at most one R, (I, J1):R, (I, J2):R  ==>  J1=J2.
```

An object does not belong to **at most one** R if, and only if, there are at least two different role fillers

```
X:nota at most one R  <=>  (X,Y):R, (X,Z):R, Y≠Z.
```

Example (contd):

```
very_simple_device isa simple_device and at most one connector.
```

□

4.2 Concrete Domains

In [Han92] restricted forms of quantification over predicates of a *concrete domain* $\mathcal{D}$ have been suggested as concept forming operators. Examples of concrete domains are Allen's temporal interval relations, rational (natural) numbers with comparison operators and real-closed fields (all of which have been implemented by CH rules). An *admissible* concrete domain has to be closed under complement (since we have to propagate the complement operator) and has to provide a satisfiability test for conjunctions of predicates. As an additional technical requirement we define the abstract domains $\mathrm{dom}_\mathcal{I}$ to be always disjoint to the concrete domain $\mathrm{dom}_\mathcal{D}$ of $\mathcal{D}$.

The syntax for the new operators in the extension TL($\mathcal{D}$) of the concept language is as follows:

every w_0 **and** ...**and** w_n **is** p
some w_0 **and** ...**and** w_n **is** p

Where w_i is of the form "R_{i0} **of** ...**of** R_{ik_i}", R_{ij} are role names, $n > 0$, $k_i \geq 0$, $i = 1, \cdots, n$, and p is an n-ary concrete predicate (constraint) of $\mathcal{D}$. These constructs are inspired by the value restriction and the exists-in restriction.

In TL($\mathcal{D}$) each interpretation $\mathcal{I}$ still interprets concept terms as subsets of $\mathrm{dom}_\mathcal{I}$. Roles link the abstract domain with the concrete domain. So, a role (resp., attribute) is interpreted as a subset of $\mathrm{dom}_\mathcal{I} \times (\mathrm{dom}_\mathcal{I} \cup \mathrm{dom}_\mathcal{D})$ (resp., as the graph of a partial function $\mathrm{dom}_\mathcal{I} \rightarrow (\mathrm{dom}_\mathcal{I} \cup \mathrm{dom}_\mathcal{D})$). The semantics of the new operators is as follows:

[2] The **at most one** construct is a restricted form of *number restrictions* [BS85].

$a \in$ (**every** w_0 **and** ...**and** w_n **is** p)$^{\mathcal{I}}$
iff, for all $b_1, \cdots, b_n \in \mathrm{dom}_{\mathcal{D}}$: $(a, b_i) \in w_i^{\mathcal{I}}$, for $i = 1, \cdots, n$, implies $(b_1, \cdots, b_n) \in p^{\mathcal{D}}$

$a \in$ (**some** w_0 **and** ...**and** w_n **is** p)$^{\mathcal{I}}$
iff, there are $b_1, \cdots, b_n \in \mathrm{dom}_{\mathcal{D}}$ such that $(a, b_i) \in w_i^{\mathcal{I}}$, for $i = 1, \cdots, n$, and $(b_1, \cdots, b_n) \in p^{\mathcal{D}}$

The denotation of $w_i^{\mathcal{I}}$ is defined inductively similar to relational product: $(a, c) \in [R$ **of** $S]^{\mathcal{I}}$ iff there exists a b such that $(a, b) \in S^{\mathcal{I}}$ and $(b, c) \in R^{\mathcal{I}}$. Reading the expressions as natural language sentences should provide a good intuition about their semantics. For a more evolved discussion the interested reader is referred to [Han92].

Since $\mathrm{dom}_{\mathcal{I}}$ and $\mathrm{dom}_{\mathcal{D}}$ are disjoint, the propagation of the complement operator **nota**, which is defined wrt $\mathrm{dom}_{\mathcal{I}}$, is more complicated in the extended formalism. We have to revise the **nota** rules for the value and the exists-in restriction:

```
I:nota every R is S <=>  I:some R is g_not S.
I:nota some R is S <=>  I:every R is g_not S.
```

Here we have introduced a global complement operator **g_not** on $(\mathrm{dom}_{\mathcal{I}} \cup \mathrm{dom}_{\mathcal{D}})^n$, $n > 0$. With R matching "w_0 **and** ...**and** w_n" the rules are also applicable to the new constructs: **every** w_0 **and** ...**and** w_n **is s** and **some** w_0 **and** ...**and** w_n **is s**, respectively. The following rules handle the global complement operator. They employ ";" to denote a disjunction in a CLP goal, which can be expanded to a collection of CLP clauses. We introduce unary predicates **concept_term**, **abstract**, and **concrete** as type constraints with the obvious meaning.

```
X:g_not g_not T  <=>   X:T.
I:g_not T.  ==>   concept_term T| I:nota T; concrete I.
(B1,···,Bn):g_not P  <=>   concrete_complement(P,Q)|
                           (B1,···,Bn):Q; abstract B1; ··· ; abstract Bn.
```

Here **concrete_complement** is a two place predicate belonging to $\mathcal{D}$ that associates each predicate of the concrete domain with its complement *wrt to* $dom_{\mathcal{D}}^n$.

Analogous to the value restriction and the exists-in restriction we have to collect (resp., generate) objects satisfying the concrete domain predicate. This can be implemented through a fixed, finite set of CH rules that collect (resp., generate) objects according to the roles and attributes occurring in the operators "from left to right" and then restrict the collected (resp., generated) objects with the concrete predicate. To simplify the presentation we give two schemata of propagation (resp., simplification) rules. For each term of the form **every** w_0 **and** ...**and** w_n **is s** that occurs in the knowledge base or in a query we introduce a rule

```
X:every w0 and ...and wn is S, {(Xij,Xi,j+1) : Rij| i = 1,···,n; j = 0,···,ki − 1; Xi1 = X}
    ==>   (X1,k1+1,···,Xn,kn+1):S.
```

Note that S is a variable that can be bound also to **g_not s** during a computation. Analogously, for a term **some** w_0 **and** ...**and** w_n **is** S we introduce a rule

```
X:some w0 and ...and wn is S
    <=>   (X1,k1+1,···,Xn,kn+1):S, {(Xij,Xi,j+1) : Rij| i = 1,···,n; j = 0,···,ki − 1; Xi1 = X}.
```

For instance, an expression **every f of r1 and r2 is p** leads to a rule `X:every f of r1 and r2 is P`, `(X,X1,1):r1`, `(X1,1,X1,2):f`, `(X,X2,1):r2 ==>  (X1,2,X2,1):P`.

As an example of a simple concrete domain we take inequalities over rational numbers. The reasoning in the concrete domain itself is implemented through the following rules which find all contradictions (but do not perform all possible simplifications).

```
X > Y <=>  Y < X.
X >= Y <=>  Y =< X.

X =< X <=>   true.
X < X <=>   false.
X =< Y <=>  number_>(X,Y)|  false.
X < Y <=>  number_>=(X,Y)|  false.
```

```
X < Y, Y < Z ==>   X < Z.
X =< Y, Y < Z ==>   X < Z.
X < Y, Y =< Z ==>   X < Z.
X =< Y, Y =< Z ==>   X =< Z.
```

The guard **number_>(X,Y)** (resp., **number_>=(X,Y)**) is true if **X** and **Y** are bound to numbers x and y and $x > y$ (resp., $x \geq y$). The predicate **concrete_complement** associating concrete predicates with their complements within the concrete domain is defined by the following facts:

```
concrete_complement(<,>=).      concrete_complement(=<,>).
concrete_complement(>=,<).      concrete_complement(>,=<).
```

The atoms which are of the form $(x, y) : \langle comparison\ operator \rangle$ or $x : (\langle comparison\ operator \rangle \langle number \rangle)$[3] and are generated by the rules for the new operator have to be translated to the infix syntax of the concrete domain:

```
(X,Y):Op <=> member(Op,[<, >, =<, >=])|  (X Op Y).
X:(Op C) <=> member(Op,[<, >, =<, >=]), number C|  (X Op C).
```

Finally, we explore the disjointness of the abstract and the concrete domain to discover contradictions:

```
X:C ==>   concept_term(C)| abstract X.
(X,Y):R ==> role R | abstract X.
(X,Y):F ==> attribute R | abstract X.
(X,Y):Op ==> member(Op,[<, >, =<, >=])|  concrete X, concrete Y.
X:(Op C) ==> member(Op,[<, >, =<, >=]), number C|  concrete X.
abstract X, concrete X <=> false.
```

Example (contd): Now we can associate price and voltage with a device and require that in an electrical configuration the voltages have to be compatible.

```
attribute price.
attribute voltage.
electrical_device isa very_simple_device and
          some voltage is > 0 and some price is > 1 .
low_cost_device isa electrical_device and every price is < 200.
high_voltage_device isa electrical_device and every voltage is > 15.
electrical_config isa simple_configuration and
          every component_1 is electrical_device and
          every component_2 is electrical_device and
          every voltage of component_1 and voltage of component_2 is >=.
```

□

The new operator can also be used to specify upper bounds. This is illustrated by a configuration where several CPUs are plugged onto a bus with the side condition that the maximal frequency of the CPUs must be less than the frequency of the bus.

```
attribute frequency.
primitive bus.
bus_device isa simple_device and bus and some frequency is > 0 .
primitive cpu.
cpu_device isa simple_device and cpu and some frequency is > 0 .
role main_device.
role sub_device.
bus_config isa configuration and
          some main_device is bus_device and every component is cpu_device and
          every frequency of main_device and frequency of sub_device is > .
```

□

[3] The latter enables numbers in concept terms.

4.3 CLP+CH(TL($\mathcal{D}$))

If we apply the CLP scheme of Höhfeld und Smolka [HS90] in a straight-forward manner to A-boxes of TL($\mathcal{D}$), we obtain a CLP language with three representation and reasoning layers.

Example (contd): The following CLP clauses specify the catalog of devices and describe possible configurations that are based on this catalog.

```
catalog(dev1) :- dev1:electrical_device, (dev1,10):voltage, (dev1,100):price.
catalog(dev2) :- dev2:electrical_device, (dev2,20):voltage, (dev2,1000):price.
possible_config(C) :-
          catalog(D1), (C,D1):component_1,
          catalog(D2), (C,D2):component_2.
```

The following queries enumerate possible configurations satisfying the requirements.

```
:-possible_config(C).
:-possible_config(C), C:electrical_config.
:-possible_config(C), C:electrical_config,
          (C,D1):component_1, D1:low_cost_device,
          (C,D2):component_2, D2:high_voltage_device.
```

The first query enumerates all possible electrical configurations comprising two devices based on the catalog, i.e., configurations comprising two **dev1**, two **dev2**, or **dev1** and **dev2**. The second query omits the configuration where **dev1** is component one and **dev2** is component two. Finally, the third query has no solution, because the catalog lists only one low-cost device and there is no high-voltage device with a compatible voltage. □

5 Conclusions

Constraint handling rules (CH rules) are a language extension for implementing user-defined constraints. Rapid prototyping of novel applications for constraint techniques is encouraged by the high level of abstraction and declarative nature of CH rules.

In this paper we investigated the terminological reasoning formalism. Flexibility was illustrated by extending the formalism and its implementation with attributes, a special quantifier and concrete domains. Applicability was illustrated by sketching a generic, hybrid knowledge base for solving configuration problems.

References

[BaHa91] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In Proceedings of the 12th International Joint Conference on Artificial Intelligence, 1991.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. Cognitive Science, 9(2):171–216, 1985.

[Coh90] J. Cohen, Constraint Logic Programming Languages, Communications of the ACM 33(7):52-68, July 1990.

[D*88] M. Dincbas et al., The Constraint Logic Programming Language CHIP, Fifth Generation Computer Systems, Tokyo, Japan, December 1988.

[Fru92] T. Frühwirth, Constraint Simplification Rules (former name for CH rules), Technical Report ECRC-92-18, ECRC Munich, Germany, July 1992 (revised version of Internal Report ECRC-LP-63, October 1991).

[Fru93] T. Frühwirth, Temporal Reasoning with Constraint Handling Rules. Technical Report Core-93-8, ECRC Munich, Germany, January 1993.

[HaJa90] S. Haridi and S. Janson, Kernel Andorra Prolog and its Computation Model. Seventh Int Conference on Logic Programming, MIT Press 1990, pp. 31-46.

[Han92] P. Hanschke. Specifying role interaction in concept languages. In Third International Conference on Principles of Knowledge Representation and Reasoning (KR '92), October 1992.

[Hol90] B. Hollunder. Hybrid inferences in KL-ONE-based knowledge representation systems. In 14th German Workshop on Artificial Intelligence (GWAI-90), volume 251, pages 38–47. Springer, 1990.

[HNS90] B. Hollunder, W. Nutt, and M. Schmidt-Schauß. Subsumption algorithms for concept description languages. In 9th European Conference on Artificial Intelligence (ECAI'90), pages 348–353. Pitman Publishing, 1990.

[HS90] M. Höhfeld and G. Smolka, Definite Relations over Constraint Languages. LILOG Report 53, IBM Deutschland, West Germany, October 1988.

[J*92] J. Jaffar et al., The CLP(R) Language and System, ACM Transactions on programming Languages and Systems, Vol.14:3, July 1992, pp. 339-395.

[JaLa87] J. Jaffar and J.-L. Lassez, Constraint Logic Programming, ACM 14th POPL 87, Munich, Germany, January 1987, pp. 111-119.

[Sar89] V. A. Saraswat, Concurrent Constraint Programming Languages, Ph.D. Dissertation, Carnegie Mellon Univ., also TR CMU-CS-89-108, 1989.

[ScSm91] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. In Journal of Artificial Intelligence, 47, 1991.

[Sha89] E. Shapiro, The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, 21(3):413-510, September 1989.

[Smo91] G. Smolka, Residuation and Guarded Rules for Constraint Logic Programming, Digital Equipment Paris Research Laboratory Research Report, France, June 1991.

[VH91] P. van Hentenryck, Constraint Logic Programming, The Knowledge Engineering Review, Vol 6:3, 1991, pp 151-194.

A Powerful Evaluation Strategy For CLP Programs

Hong Gao
Department of Computer Science
State University of New York
Stony Brook, NY 11794
gaohong@cs.sunysb.edu

David S. Warren
Department of Computer Science
State University of New York
Stony Brook, NY 11794
warren@cs.sunysb.edu

Abstract

This paper presents a new, powerful evaluation strategy (OLDTC-AM) for CLP programs. OLDTC-AM is developed by combining the OLDT evaluation strategy with a logical answer manipulation mechanism. Under the OLDTC-AM evaluation strategy, the termination characteristics of CLP programs are greatly improved and the expressive abilities of CLP languages are greatly increased. One application of this power is the direct solving of optimisation problems. Through an example, we show how an optimisation problem can be expressed as a CLP program simply and be solved logically in the constraint logic framework.

1 Introduction

To review briefly, CLP is a framework for constraint handling in logic programming [4] [5]. In other words, CLP is a scheme which extends Horn clauses with constraint predicates[1]. The CLP scheme defines a class of languages, in which each instance, CLP($\mathcal{D}$), is a programming language and is obtained by the specification of the structure of a computation domain $\mathcal{D}$. For example, CLP($\mathcal{R}$) is a logic programming language with the computation domain being the real numbers $\mathcal{R}$.

Prolog is a logic programming language based on Horn clauses. The evaluation strategy that Prolog uses is the SLD proof strategy for Horn clauses. Formally SLD resolution is refutation complete for Horn clauses [7]. This means that if a ground answer is a logical consequence of a program, then there is an SLD refutation that generates that answer (or a more general one). However, there may be some paths in the tree of refutation that are infinite in length. Since Prolog must search this tree for answers, its depth-first search may get caught on an infinite path, before it gets to an answer.

OLDT, as an alternative evaluation strategy for Horn clauses proposed in [1] [10], is complete and terminates in many cases for which Prolog's strategy loops infinitely.

Since CLP extends Horn clauses with constraint predicates, in most current CLP systems a CLP program is evaluated in the following way: for an ordinary predicate, the SLD refutation strategy is used to perform inference; for a constraint predicate, the constraint solver is called to solve that constraint. The SLD evaluation part in CLP's underlying engine causes the same termination problem as Prolog has.

Inspired by the way OLDT works for Prolog programs, we propose to replace SLD with OLDT in current CLP evaluation strategy, and further develop an answer manipulation mechanism. The new evaluation strategy is called OLDTC-AM (*OLDT with Constraint handling and Answer Manipulation*).

In the following sections, we first describe the OLDTC-AM evaluation strategy through two steps: OLDTC based on OLDT and AM (answer manipulation). Then we discuss the powerful strengths OLDTC-AM has as an evaluation strategy for CLP programs from three aspects. Finally we conclude the major results we present in this paper.

[1] We do not discuss logic programs with negation in this paper.

2 OLDTC-AM

In this section, we briefly introduce the OLDT evaluation strategy. Based on that we explain how OLDTC-AM works for CLP programs.

2.1 OLDT

OLDT stands for *Ordered selection strategy with Linear resolution for Definite clauses with Tabling*. The concept underlying it is termed "memoing". In a deterministic language, the idea of "memoing" is simple: during execution, maintain a table of procedure calls and the values they return; if the same call is made later in the computation, do not re-execute the procedure, but use the saved answer to update the current state as though the procedure had been called and had returned that value.

Conceptually, Prolog's nondeterministic computation can be transformed into a set of deterministic ones in the following natural way [11]: a machine that is carrying out a nondeterministic procedure is viewed as duplicating itself at a point of choice, as disappearing when it encounters failure. So at any time there is a set of deterministic machines computing away. The set gets larger when any one has to make a nondeterministic choice, and it gets smaller when any one fails. To add memoing, a single global table, which contains every procedure call that has been made by any machine and the answers that have been returned for each such call, is maintained. Since the computation is nondeterministic, there may be none, one, or many answers for any single call in the table.

For example, there is a simple Prolog program P as follows.

```
p(X,Y) :- arc(X,Y).
p(X,Y) :- arc(X,Z), p(Z,Y).
arc(a,b). arc(b,a). arc(b,d).
```

Its evaluation procedure in corresponding SLD and OLDT is shown in Figure 1.

2.2 OLDTC

OLDTC is very similar to OLDT except that it includes constraint handling. During computation, if a machine encounters a constraint predicate (i.e., the predicate for a primitive constraint), it calls a constraint solver[2] to solve this constraint instead of treating this predicate as an ordinary predicate and applying the OLDT algorithm to it. If the constraint is a linear equation or inequality, it is solved directly. If it is a non-linear constraint, it will be delayed and may get solved later if it becomes linear due to the solving of other constraints. Hence on each machine, the conjunction of constraints changes during computation, with constraints being added and some being simplified away.

In general, OLDTC works as follows. When a machine encounters a call, it looks the call in the table. If it is not there, it adds the call (as in OLDT, but without constraints) and makes the call passing no constraints into it. When a machine returns, it adds its answer and its simplified constraints to the table. If when a call is made, a matching call is found in the table, machines are forked off for each associated answer (and corresponding set of constraints) in the table. The constraints in the table for the answer are conjoined in with the current constraints of the calling machine. Note in this way, since constraints are not passed down into subroutines, constraints are completely computed bottom-up.

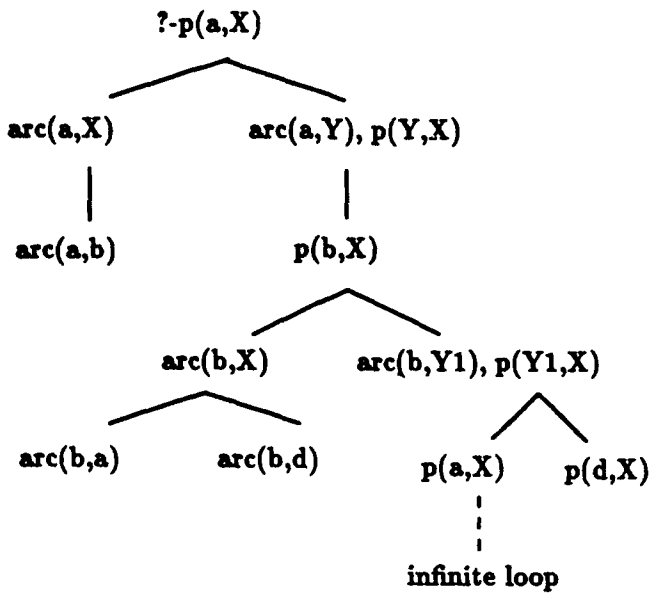
Since we can consider the answer substitution to be a set of equality constraints, answers saved in the table entry for each CLP program predicate call can be viewed as a set of simplified, consistent constraints.

2.3 Answer Manipulation

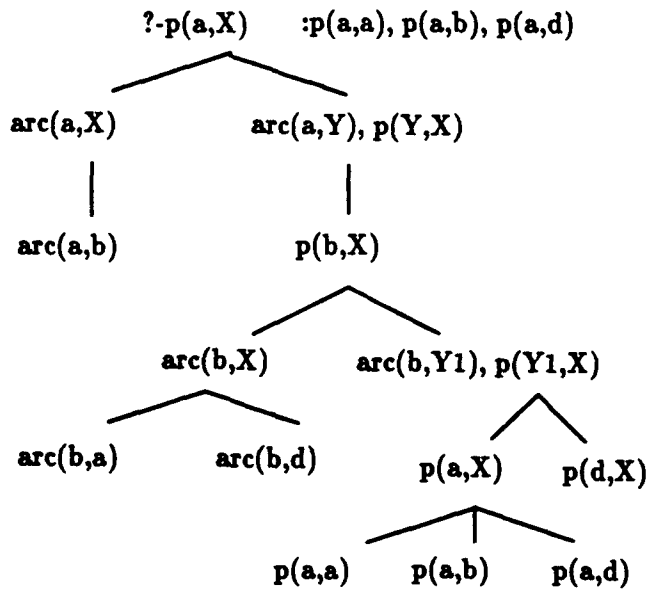
For the answers saved in the table entry, what are their logical meanings? One way to think about the table entries is that they represent rules that logically follow from the program and the constraint theory. For example, if a table entry for a program P is:

$$[\, call : p(X_1, X_2),\ [ret : [X_1 = A, X_2 > b], ret : [X_1 = f(X_3), X_3 = c, X_2 > b]]\,]$$

[2] Based on a concrete computation domain $\mathcal{D}$.



SLD Evaluation Strategy



OLDT Evaluation Strategy

Figure 1: SLD and OLDT evaluation strategy

then this means that the following rules are logically implied by the program P.

$$p(X_1, X_2) :- X_1 = A,\ X_2 > b.$$
$$p(X_1, X_2) :- X_1 = f(X_3),\ X_2 > b,\ X_3 = c.$$

Based upon the logical meaning of answers, we can define two logic operations on answers, one is *answer projection*, the other is *answer merging* [2].

2.3.1 Answer Projection

By definition an answer to a goal should be bindings to the variables appearing in the goal itself. If after a computation, an answer for a goal G contains variables which do not appear in G, then a logically equivalent form which contains only bindings to the variables in G would better be output. This can be done through an operation called *answer projection*.

There is a variety of ways of doing answer projection. One way we propose here is the use of Two-Phase Simplex algorithm. Two-Phase Simplex algorithm is an algorithm developed in the operation research field [8] [9]. Its main function is: given a set of linear constraints

$$a_{11}X_1 + \cdots + a_{1m}X_m \leq b_1$$
$$\cdots\cdots$$
$$a_{n1}X_1 + \cdots + a_{nm}X_m \leq b_n$$
$$X_1 \geq 0$$
$$\cdots\cdots$$
$$X_m \geq 0$$

and an objective function,

$$C = X_1 + \cdots + X_m$$

Two-Phase Simplex algorithm returns the maximal value for C.

Now let us look at how Two-Phase Simplex algorithm can be employed in an answer projection procedure through an example. Suppose during a CLP program computation, we have a table entry like

$$[\,call : p(X),\ [ret : [X < U + V,\ 2 * U + 3 * V < 10,\ U + 5 * V < 20,\ U \geq 0,\ V \geq 0]]\,] \quad (1)$$

This entry tells us that the following rule is implied by the original program.

```
p(X) :-
    X < U+V,
    2*U + 3*V < 10,
    U + 5*V < 20,
    U ≥ 0,
    V ≥ 0.
```

The above rule says that $p(X)$ is true as long as X is less than any sum of value U and V, where U and V are subject to the following linear constraints

```
2*U + 3*V < 10,
U + 5*V < 20,
U ≥ 0,
V ≥ 0
```

Imagine that if we can find the maximal value of the sum of U and V, for example σ, then logically $X < \sigma$ is an equivalent answer to the complicated one in (1).

By regarding $C = U + V$ as an objective function constrained by the above four linear constraints, we can see that finding the maximal value of $C = U + V$ is a typical problem in the operation research field. By applying Two-Phase Simplex algorithm to this problem, we can obtain the maximal value of $C = U + V$,

say σ. So we can return the simpler expression $X < \sigma$ which eliminates the non-goal variables U and V as the answer to $p(X)$.

For most current CLP systems, answer projection is only performed at the last step, i.e., when CLP systems return final answers to the initial query[3]. The key point here is that we propose to perform answer projection even in the middle of a computation. This is impossible for evaluation strategies which do not have memoing mechanism. Without memoing mechanism, there is no way to tell which variable does appear in the predicate call and which does not. In an evaluation strategy with memoing mechanism, such as OLDTC, adding answer projection will simplify answers saved in the table entry for a predicate call, and hence reduce the computation complexity when later this answer is used for the same predicate call.

Notice that in Jaffar's CLP($\mathcal{R}$) system, the Phase I of the Two-Phase Simplex algorithm is used, but only for consistency checking of a set of collected constraints. Here the full phase of Simplex algorithm is proposed to perform answer projection.

2.3.2 Answer Merging

We know answer projection works on a single answer of a predicate call. Due to the nondeterministic feature of logic programs, there may have several answers computed for a predicate call. By studying the inter-relationships among these answers, we propose that further answer simplification can be done through an operation called *answer merging*.

Let P be a logic program and G be a goal. Let A_1 and A_2 be two answers of G. If $A_1 \leftarrow A_2$ is logically implied by P, then A_2 is called a *redundant answer* compared with A_1. The reason for calling an answer redundant is: if A_1 is an answer for G in program P, then $G \leftarrow A_1$ has been proved from the program P. So knowing that $A_1 \leftarrow A_2$, we can prove $G \leftarrow A_2$ immediately. In other words, $G \leftarrow A_2$ can be inferred automatically from $G \leftarrow A_1$ and $A_1 \leftarrow A_2$. Therefore A_2 is redundant. The elimination of redundant answers is performed through answer merging.

For example, suppose we have two answers for the goal $p(X)$ as follows:

$$[\, call : p(X),\ [ret : [X > 6],\ ret : [X > 10]]\,]$$

This means that the following rules are implied by the original program.

```
p(X) :- X > 6.
p(X) :- X > 10.
```

Since $X > 10$ implies that $X > 6$, the second rule can be inferred automatically through the first rule and the relation between $X > 10$ and $X > 6$. Therefore we regard $X > 10$ as a redundant answer and eliminate it from the table entry.

As we discussed before in answer projection, answer merging is also impossible to implement in evaluation strategies which do not have memoing mechanism. Without memoing mechanism, computed answers are not saved, hence comparisons among different answers cannot be performed and their mutual implying relations cannot be determined.

By combining OLDTC with the above two answer manipulation mechanism together, we develop a new evaluation strategy for CLP programs. It is OLDTC-AM.

3 Advantages of OLDTC-AM

Using OLDTC-AM as the new evaluation strategy for CLP programs has the following advantages.

3.1 Efficient Computation

OLDTC-AM increases the efficiency of computation. First during a computation, if the same call is encountered later, by the principle of OLDT, the same call is not recomputed, instead all the answers saved

[3] Note that different ways may be used to do answer projection.

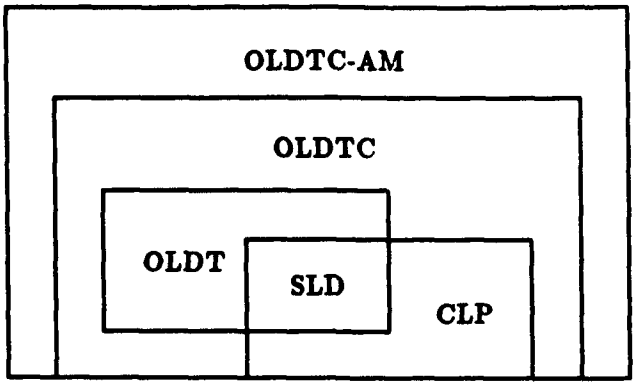


Figure 2: Computational ability comparison among various evaluation strategies

in the table for this call are used to update environments as through the call is computed. Second, answer projection simplifies answers saved in the table entry, so the computation becomes less complex by using simpler answers if the same predicate call is encountered again. Third, answer merging prunes the search space by eliminating redundant answers from the set of answers to a call. So only the non-redundant answers are used in the computation.

3.2 Termination Characteristics

OLDTC-AM makes more logically-meaningful CLP programs terminate. In other words, it makes more logically-meaningful CLP programs computable. The computational abilities among various evaluation strategies: SLD, OLDT, CLP[4], OLDTC, OLDTC-AM, can be summarised in Figure 2.

In Figure 2, each rectangle represents a set of computable logic programs and the bigger rectangle computes more logic programs than the smaller one does. We already know that OLDT makes more logically-meaningful programs computable than SLD. We also know that current CLP evaluation strategy, such as the one used in CLP($\mathcal{R}$), has constraint handling ability SLD does not have. Hence current CLP evaluation strategy computes more logic programs than SLD. It is obvious that OLDTC computes more logic programs than separate OLDT or current CLP evaluation strategy due to its combined power of OLDT and constraint handling. Compared with OLDTC, OLDTC-AM further terminates more logic programs by having answer manipulation mechanism. This can be understood through the following example.

We know that OLDTC employs a memoing mechanism to terminate more CLP programs. But for the following CLP program

```
p(X,Y,C) :-
    C ≥ W,
    q(X,Y,W).
p(X,Y,C) :-
    C ≥ W+C1,
    q(X,Z,W), p(Z,Y,C1).
q(a,b,1). q(b,a,2). q(b,d,4).
```

and the query `p(a,d,C)`, the computation in OLDTC still goes into infinite loop. There are infinitely many answers saved in the OLDTC table for call `?-p(a,d,C)` due to the existence of loop between a and b.

$$[\, call : p(a, d, C),\ [ret : C \geq 5,\ ret : C \geq 7,\ ret : C \geq 9, \cdots]\,]$$

But if the above program is evaluated in OLDTC-AM, since any answer (except $C \geq 5$) in the above list are redundant compared with $C \geq 5$, it is eliminated. Only $C \geq 5$ is saved in the table entry. So the computation terminates.

[4] Those used in current CLP systems, i.e., SLD inference engine with constraint handling.

3.3 Optimization Problems

Last but not least, OLDTC-AM further enhances the expressive power of CLP languages in the sense that optimisation problems can be solved simply and logically in the constraint logic framework.

An example of optimisation problems is the "shortest path" problem. We will represent a weighted directed graph by a set of facts of the form:

$$arc(Source_node, Target_node, Weight)$$

Then we can define a program as follows:

```
suboptimal_length(X,Y,C) :-
    C > W,
    arc(X,Y,W).
suboptimal_length(X,Y,C) :-
    C > W+C1,
    arc(X,Z,W),
    suboptimal_length(Z,Y,C1).
```

which says that

- (first rule): a path from X to Y is too long if its length is longer than the weight of an arc from X to Y;
- (second rule): a path from X to Y is too long if its length is longer than the sum of the length of an arc from X to Z and the length of going from Z (Z as an intermediate node) to Y.

During evaluation by OLDTC-AM, the query `?- suboptimal_length(a,d,C)` may compute a table entry like:

$$[call : suboptimal_length(a, d, C), ret : [C > 24]\,]$$

This corresponds to having computed the following rule:

$$suboptimal_length(a, d, C) : -C > 24.$$

which means that we have proved that from a to d, the path whose length is longer than 24 is a suboptimal one. Now say there is a new answer computed for this call:

$$ret : [C > 36]$$

This tells us that another rule is implied by the program:

$$suboptimal_length(a, d, C) : -C > 36.$$

But since $(C > 24) \leftarrow (C > 36)$, i.e., $(C > 36)$ implies $(C > 24)$, the latter rule logically follows from the previous rule in the constraint theory of $>$. Therefore, we need not add this new but redundant answer to the table, but can just fail it according to the definition of answer merging. When the computation is finished, the table entry will have an answer, for example, such as:

$$suboptimal_length(a, d, C) : -C > 16.$$

which indicates that from a to d, any path whose length is longer than 16 is suboptimal. In other words, any path from node a to node d longer than 16 is too long and not an optimal one. Therefore, the optimal (or shortest) length of a path from node a to d should be 16.

In solving optimisation problems, OLDTC-AM has its advantages over the SLD resolution strategy used in most current CLP systems [3] [6]. An optimisation problem is a generate-and-test problem which needs to search the whole derivation space and compare all feasible solutions there. But in current CLP systems

(such as Jaffar's CLP($\mathcal{R}$) system), once a branch in the derivation tree succeeds, substitutions along this branch are returned as an answer to the programmer. The systems continue to search other answers upon the programmer's request. Logically, this sequence of answers is disjunctive, but current CLP systems never do any comparisons among these disjunctive answers. So if a programmer wants the best answer among all solutions, he must store all solutions in a data structure and compare them one by one. All these processes and the control strategy of answer searching have to be hand-coded explicitly into his program. In this way, the resulting program will become complicated and lack an intuitive logical meaning when specifying a practical optimisation problem. But with the OLDTC-AM resolution strategy, all the above data structures and the control scheme are left to the underlying system – memoing and answer merging. With memoing, the system saves the computed solutions automatically. With answer merging, the system compares saved disjunctive solutions and gets rid of those redundant (or suboptimal) solutions. Therefore a programmer can write much purer logic programs without having to code the search strategy into his programs.

4 Conclusion

The goal of this paper has been to present a powerful evaluation strategy for constraint logic programs. In order to do this, we employ the main ideas of the OLDT evaluation strategy, develop a logical answer manipulation mechanism, and combine them together to form OLDTC-AM. Due to the memoing and answer manipulation mechanism, OLDTC-AM terminates more CLP programs than the SLD evaluation strategy underlying most current CLP system engines. In the meanwhile, OLDTC-AM improves the efficiency of computation by eliminating redundant answers and redundant recomputations. With OLDTC-AM, an optimisation problem, that would otherwise requires an ad hoc algorithm, can be easily specified as a constraint logic program and can be solved logically in the constraint logic framework.

References

[1] Susanne Wagner Dietrich and D.S. Warren. Extension Tables: Memo Relations in Logic Programming. Technical report, Computer Science Department, SUNY at Stony Brook, March 1986.

[2] Hong Gao. *Declarative Picture Description and Interpretation in Logic.* PhD thesis, Computer Science Department, SUNY at Stony Brook, 1992.

[3] N.C. Heinze, J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP(R) Programmer's Manual, version 1.1. Technical report, I.B.M T.J. Watson Research Center, November 1991.

[4] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM Principles of Programming Languages Conf.*, pages 111–119, January 1987.

[5] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Proc. of the 4th International Conf. on Logic Programming*, pages 196–218, May 1987.

[6] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP(R) Language and System. Technical Report, April 1988.

[7] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[8] Claude McMillan. *Mathematical Programming: An Introduction to the Design and Application of Optimal Decision Machines.* Wiley, New York, 1970.

[9] Katta G. Murty. *Linear and Combinatorial Programming.* Wiley, New York, 1976.

[10] Hisao Tamaki and Taisuke Sato. OLD Resolution with Tabulation. In *Proc. of the 3rd International Conf. on Logic Programming*, pages 84–98, 1986.

[11] David S. Warren. Memoing for Logic Programming. *Communications of the ACM*, pages 93–111, March 1992.

Practical Issues in Graphical Constraints

Michael Gleicher
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
gleicher@cs.cmu.edu

Abstract

Use of constraint-based techniques in interactive graphics applications poses a variety of unique challenges to system implementors. This paper begins by describing how interface concerns create demands on interactive, constraint-based, graphical applications. We will discuss why such applications must be able to handle systems of non-linear constraints, and survey some of the techniques available to solve them. Employing these numerical algorithms in the contexts of interactive systems provides a set of challenges, including dynamically setting up the equations to be solved and achieving adequate performance and scalability. This paper will explore these issues and describe the methods we have used in our efforts to address them.

1 Introduction

The ability to represent and maintain relationships among objects can be an extremely useful tool in a graphical application. Since the earliest interactive graphical applications[25], the use of such constraint techniques has been demonstrated in applications including drawing, 3d modeling, user interface construction, animation, and design.

In employing constraint-based techniques in such graphical applications, system designers must face a variety of new challenges. This paper aims to describe some of these challenges, and discuss some techniques to address them. We begin by looking how usability concerns for such applications create demands on what systems must do. We will discuss why these applications will often demand the power and generality of non-linear numerical techniques. Issues in employing such algorithms within interactive systems will be surveyed, with an emphasis on how the equations to be solved can be set up and how adequate scalability and performance might be achieved.

For this paper, we are concerned with the class of graphical applications where the user creates and edits models made up of a number of graphical objects. An example is an object-oriented drawing program, where the model (or drawing in this case) is made up of lines and circles, but not a painting program in which the model is a bitmap or image. Also, the constraints that we are concerned with in this paper are those used within the model, for example, to enforce a relationship within a drawing. This is different from the common use of constraints in user interface construction, where a constraint is used by the programmer to enforce internal consistency within the program. These two views of constraints provided different sets of challenges, although some of the issues and solutions presented here apply to both.

2 The Challenges of Constraint-Based Graphical Applications

An interactive graphical application with constraints must contend with the same basic challenges as those without constraints. However, there are challenges which are inherent when constraints among objects are provided. Constraints obviously add to models, in addition to the graphical objects, constrained models also contain constraints which must be stored, displayed, edited, saved, etc. More significantly, constraints change the nature of interaction in a graphical application. Without them, actions only affect the objects to which they refer. For example, dragging an object moves only the object. With constraints, this locality is lost: altering one object may cause other objects to be affected. This global nature of constraint operations is at the core of many of the difficult issues in employing constraints. It introduces challenges in implementation, in performance, and in usability. The latter is potentially most concerning, not only for its difficulty, but also because usability concerns create further challenges for implementation and performance.

Without user specified constraints, graphical objects have fixed behaviors. For instance, an ellipse in a drawing program behaves like an ellipse. The system designer can design a good, usable behavior which the user can learn and apply to all ellipses. When user specified constraints among objects are introduced, the situation changes. To begin with, the behaviors can become more complicated because of interactions among objects. Each combination of objects and constraints will have its own behavior. These behaviors are specified by the user in terms of the constraints; the user is effectively programming.

As in more traditional programming, complexity in the constrained behavior of a graphical model becomes a problem when it has bugs, e.g. when the behavior isn't what is desired or expected. The most obvious form of bug is when the constraints force the model into a configuration which is not what the user desires, or the constraints prevent the user from achieving a desired configuration. Another class of constraint bug stems from bad constraints where solutions cannot be found, either because of conflicting specifications or solver failures.

Because constraint errors occur, interactive graphical applications which provide constraints to users must deal gracefully with bad situations, such as conflicting or redundant constraints. Underdetermined models also must be handled, as it is impractical to expect the user to fully specify all possible degrees of freedom. Because of the potential for errors, it is crucial to aid the user in understanding the complex behaviors of constrained models. For this task, providing continuous motion animation seems to be key. This places demands on systems to provide rapid enough iterations to provide the illusion of continuous motion. Another important weapon in avoiding constraint bugs is the development of specification techniques which help avoid them; this is evidenced by the large effort in automatically inferring constraints, such as [10, 1, 16].

The interactive nature of constraint-based graphical application also causes the systems of constraints to be dynamic. Typically, as the user edits the model, constraints are added, removed, and altered. While there are some applications, such as [23], where it is possible to separate manipulation and modeling, applications must typically interleave altering the constraints with solving them. The ability to rapidly alter the set of constraints can also be used to create new facilities in the solver. For example, switching a constraint on and off at the correct times permits the creation of inequality constraints, using what are called active-set methods[4].

With all of the discussion of constraints, it is easy to lose sight of the fact that constraints are usually a tool to aid in the process of creating graphical models. This leads to demands of reliability and robustness for solvers. Transparency also means that users should not be forced to deal with

equations. Artifacts of the solving process must be hidden from the user. For example, users should not be forced to create constraints which have properties which stem only from solver limitations; for example, some solvers require constraints to be expressible as directed acyclic graphs.

2.1 An Example Application

A constraint-based drawing program demonstrates how the issues in building interactive constraint-based graphical applications manifest themselves. The idea of using constraints in a drawing program is not new; in fact, it dates back to one of the earliest interactive systems[25]. However, despite the nearly universal agreement on their utility, constraints never really caught on in graphical applications. What has been successful are direct manipulation programs.

We have built a drawing program called *Briar*[10, 5] which aimed to keep the the features of the successful direct manipulation systems, but to augment them with constraints[6]. Briar is based on an existing, highly evolved direct manipulation drawing techniques[2], and augments them by making the relationships between objects persistent. Briar provides a set of snap-dragging features to help users draw precisely and quickly. However, unlike other snap-dragging systems, Briar provides the facility of making these snapping operations into persistent constraints. This can be done without the user explicitly specifying the constraints. Briar uses a visual language for displaying the constraints, which closely parallels the snap specifications, and also provides simple methods for deleting constraints based on drawing operations. The avoidance of direct reference to constraints avoids several classes of constraint bugs including conflicts[6].

Despite its constraint features, Briar maintains the fundamental direct manipulation feel of dragging. Like more traditional programs, objects are dragged and move with continuous motion, except that in Briar, constraints among the objects can be maintained. For example, the user can draw a mechanical contraption, and have it stay together when the crank is turned.

Briar only represents two types of constraints: point—on—object and point—on—point. More complicated relationships, such as distance, orientation, or co-linearity, are created by combining these simple elements with special alignment objects. These more complex relationships lead to non-linear equations which Briar's solver must handle. Hierarchical grouping with rotation and interesting objects also lead to non-linearities.

Another important aspect of the constraints in Briar is that they are dynamic. The user is continually creating and destroying constraints. These changes occur during drawing operations, so it would be unacceptable if adding or deleting a constraint were time consuming.

3 The Need for General Purpose Solving

The interface needs of interactive graphical applications place difficult performance demands on constraint algorithms. In order to keep up with interactive rates, it is tempting to make restrictions on the types of constraints which the solver can handle. However, many graphical applications require maintaining sets of non-linear equations, and the generality given by the ability to manage this general class of equations affords interesting possibilities in systems.

Even in the most basic 2d applications, non-linear equations arise. Simple geometric relationships, such as distance and orientation, give rise to non-linear equations. Many graphical objects are most easily represented in ways that give rise to non-linear functions. Similarly in 3D, many of the interesting relationships among objects require non-linear equations.

When the realm of possible constraints is expanded to the class of non-linear equations, there is more flexibility to devise interesting constraints. For example, we have placed constraints on the outcome of viewing transformations[11], on the positions of reflections, and on the results of lighting calculations, permitting constraining the color that an object appears. Such constraints are important as they permit users to control models directly in terms of aspects that they are interested in.

4 Solving Constraints

A constraint in an interactive graphical application is a relationship, typically geometric, among objects. A constraint is represented by an equation which must hold for the constraint to be satisfied. This equation is over the variables which determine the configuration of the model, called the *state vector.* The standard form of these equations is to write them as some function of the state vector equals a constant, often zero with no loss of generality. Inequalities are similarly handled by an equation which states that the function is greater than zero. This function is called the *constraint function.* The job of a constraint solver is to find configurations of the state vector for which the constraints hold. If continuous motion is to be achieved, the solver must be called for each frame.

A guaranteed general method for solving systems of non-linear equations does not exist, and there are arguments that such a method cannot exist[21]. However, there are methods, which despite their lack of total generality and guarantees, perform reasonably on realistic problems. These methods iteratively converge on a solution to the equations. A well known iterative method for solving non-linear systems is Newton's method, which has many variants and is the basis for many of the more sophisticated techniques. For each iteration, these methods solve a linear system to compute the next value.

We have been using a variant of these iterative approaches in our work, which we call *differential methods.* Rather than specifying what the desired values for constraint functions are, we instead specify how they are to change, e.g. their time derivatives. For example, we might specify that something is not to change, or is to move towards a target. Such an approach is useful for interaction because we want our objects to move continuously, rather than jump to their goal positions. The derivatives of variables can be computed from the derivatives of the constraint functions by solving a system of linear equations, even if the constraint functions themselves are non-linear. Differential methods are detailed in [11] and [8].

Differential methods and Newton-like methods are very similar. Both repeatedly solve linear systems to iteratively move towards satisfying non-linear equations. One way to describe the difference might be that a Newton-style method attempts to race towards the goal as fast as possible, regardless of the route taken, while the differential methods try to take a good continuous route, even if it takes longer. The former method runs more of the risk of speeding off in a wrong direction, but may arrive at its goal sooner.

The method used to solve the constraints is not really important. What matters is that the constraints are solved in a manner that is fast, robust and reliable. In fact, precision, another typical concern of numerical analysis, is typically not as important — we are often willing to let our objects be an imperceptible tenth of a pixel apart if it allows our solver to be faster. With an iterative method, we gain control over this tradeoff, as we can stop iterating when the algorithm has gotten close enough. Iterative methods can be used not only for the non-linear system solving, but also for solving the linear systems.

4.1 Solving Linear Systems

Almost any method chosen will repeatedly solve systems of linear equations based on the derivatives of the non-linear equations.[1] Linear system solving dominates the computational complexity of constraint-based graphical applications (see section 5), and is a key place where stability and reliability concerns must be met.

The linear system solver at the core of the constraint solver must be able to handle ill-conditioned and singular cases. To better handle these cases, we use a variant of damping, a technique seen in robotics [26, 19] and in the Levenberg-Markardt method [21]. Such methods add small amounts to the diagonal elements of the matrix raising their condition number and permitting them to be solved more easily. The method effectively trades some precision in the constraint calculation for better performance and stability.

To solve linear systems in our constraint applications, we have used a conjugate-gradient algorithm. This algorithm is particularly attractive because it permit exploiting sparsity without pre-analysis. Because it is an iterative technique, we have the ability to trade precision for performance by adjusting tolerance parameters. While there are techniques, such as singular value decomposition, which better handle singular and near-singular matrices, these techniques do not exploit sparsity as easily nor permit the performance adjustments. In [22], the tradeoffs between SVD and conjugate gradient are explored more closely.

5 Scalability and Performance

Performance is important to interactive graphical applications so they can achieve the appearance of *continuous motion*. In a conventional drawing program this can be achieved easily as only one object is moving at a time. If it is a complicated or compound object, it can be drawn in a simpler form, such as a bounding rectangle, because it cannot change internally. Because of this constant $O(1)$ complexity in the interactive loop, direct manipulation drawing is practical on small computers.

In a constraint-based system, the constant time portions of drawing systems no longer exist. Many objects can potentially change at once. Where this complexity hurts the most is in constraint solving, but the fact that many object move simultaneously also makes other things, like redraw, more complicated. Multiple objects moving also raises the cognitive complexity of drawing, as the behaviors become quite complicated.

The computational complexity of constraint-based graphical applications is dominated by the linear systems which must be solved in order to solve the non-linear equations. Solving a system of linear equations is, in the most general case, an $O(n^3)$ problem. However, because the matrices which arise in graphical constraint problems are typically sparse, the complexity can be lower. Because each constraint only affects at most a small constant number of objects, the matrices only have $O(n)$ entries in them, and can therefore be solved in $O(n^2)$ time[22]. For certain classes of constraint problems, the linear system can be solved in linear time[24].

To maintain interactive performance, it is critical to reduce the complexity of solving algorithm by exploiting the sparsity of the systems which are solved. However, without severely restricting the class of models which the user can build, this still leaves greater than linear complexity. Since

[1] There are non-global methods for solving non-linear equations which do not solve linear systems. In [20] these *relaxation* or *penalty* methods are described, including a discussion of why they are not good.

we cannot reduce the polynomial coefficient, one must instead reduce the size of the problems which are solved, without imposing size restrictions on the user. Tactics available for this include partitioning the constraints into smaller subproblems, which is explored in [22], and removing "dead" objects and constraint from the computations.

The basic strategy for reducing problem size is to determine which objects might move and only operate on these objects. Once the set of objects is pared, constraints which depend only on dead objects can be removed. Constraints which depend on both live and dead variables will only alter the live ones. Information about whether or not an object is alive can be used for other purposes, such as snap-target pruning[10], or simplifying displays.

There are many sources of objects to remove from the constraint system to reduce the size of the problems which must be solved. One obvious source is the user, who, for example, might want to freeze an object. It is also useful to cluster variables and allow the user to select whether these classes should be enabled or not. For example, in a 3d modeling system, the constraint solver might be used to control lighting, position the camera, and shape object geometry. However, the user will often only want to control one of these at a time. It is therefore useful to allow disabling entire classes of objects. This is especially important with constraints such as through-the-lens camera controls[11] which can affect many different types of variables.

The are two reasons to automatically disable an object or variable. The constraints might completely restrict an object (or a variable) from changing, or the object might not be connected to anything which might cause it to move, such as the mouse. For these two questions, exact answers are unavailable in general. It might require proving an arbitrarily hard geometric theorem, or doing expensive numerical calculations. However, in practice one can remove many objects, although it is difficult to guarantee the smallest possible set of active variables. In our systems, we have used techniques such as dependency analysis and simple geometric theorem proving.

Freezing variables by removing them from the global state vector is what we call a *cheap constraint*, that is, a constraint which can be enforced without adding to the set of equations which must be solved. Notice that unlike most constraints, freezing will make things run faster since it removes variables, rather than adding constraints. In certain cases, it is possible to translate other constraints into freezes, for example, a constraint which nails a point on the object might be expressible as a freeze if the point is computed directly from the values of variables. Another type of cheap constraint can be created for equating two variables by having them share a single location in the global state vector. This notion of "merging" is similar to that in [3].

6 Formulating Equations

Most of the methods for solving systems non-linear equations described in section 4 have similar requirements for what the equations need to provide. They need to be able to evaluate the constraint functions and their derivatives to form the linear systems which are solved. These evaluations must be able to take advantage of sparsity in the resulting derivative matrices to achieve needed performance in evaluations. Since these functions are defined in response to dynamic creation and destruction of constraints, the creation of the functions themselves must be dynamic.

Constraint-based applications must be able to dynamically define functions. They need to be able to rapidly evaluate these functions and their derivatives. These functions are built by composing other functions together. At a low level, one might consider building functions out of basic mathematical primitives such as addition; however, this composition occurs at higher levels

of abstraction in constraint systems too. For example, constraint are typically defined in terms of points on objects. It is then the job of the objects to compute these point positions in terms of their state variables. This provides an important layer of modularity: the constraints can be defined independently of the objects.

A function can be represented as a directed acyclic graph[2] with composed functions at the nodes, and arcs representing composition. Our approach to providing function composition in the dynamic setting of interactive applications is to provide a tools for managing these function graph structures. In *Snap-Together Math,* function elements are "wired" together to make more complicated functions.

Evaluating the values and derivatives of an expression involves traversing the graph. To compute a value, a node requests the values from its predecessors and then performs its local function on these results. The chain rule for derivatives provides a similar method for computing derivatives. To compute the derivative of a node with respect to the global inputs, a node asks its predecessors for their derivatives, and multiplies these intermediate results by the derivative of its internal derivative matrix. This process is called *automatic differentiation,* and is superior to building the global derivative matrix symbolically in most situations[13, 15].

At the leaves of the function graph are the variables over which the function is computed, the state vector of the model. The state vector contains the parameters of the graphical objects which make up the model. The state of the system is distributed among objects, however, numerical algorithms require state to be gathered into a single global vector. The positions of variables in this vector are significant as they determine which columns of the derivative matrices correspond to which variables. It would be possible to keep variables in a large vector and have the objects simply look in this larger structure for values. Such an approach, however, violates encapsulation of objects and makes it more difficult to switch variables on and off. In our approach, variables from object state vectors are gathered into a larger vector when needed. By selecting which variables are gathered, it is simple and fast to switch among sets of variables.

We have implemented function composition and automatic differentiation in an object-oriented tool called *Snap-Together Math.* Rather than requiring special graph node objects, it allows application objects to mix in the ability to "speak mathematics." This permits application objects to participate directly in calculations. They must only respond to a simple protocol. This simplifies applications by reducing the need for special math objects which must be allocated and maintained. Snap-Together Mathematics uses a specially designed sparse matrix representation and does extensive caching. Snap Together Math is described in detail in [12], and a previous version is described in [9].

7 Putting it Together

A wide variety of graphical applications might employ constraints. Any of these applications will face the same issues previously described. Fortunately, the solutions proposed are general enough to apply across many applications, and can be encapsulated into a toolkit to support a variety of applications.

At a low level, any application which employs numerical constraint will gain leverage from a library of mathematical structures and algorithms. Our mathematical toolkit provides support for such things as vectors, matrices, and differential equations in an object oriented manner. It includes

[2]It is not a tree as common subexpressions might be shared.

several varieties of sparse matrices, and a variety of linear system, non-linear system, and ordinary differential equation solvers.

Snap-Together Math, described in section 6 is built on top of the mathematical library. In addition to the support for dynamically composing and rapidly evaluating functions, it also includes interfaces differential and Newton-style solvers. Many applications, including Briar, have been built with these tools.

The Bramble graphics toolkit, built on top of Snap-Together Math, aims to provide a framework for building graphical applications with constraints. Previous toolkits, such as Garnet[18], Thinglab II[17], and Mel[14], employ constraints to aid programmers in application development. Bramble, in contrast, primarily aims to provide constraints as for user level services. Although, it does appear that the the differential constraint techniques provided in Bramble simplify the task of building graphical applications by helping separate manipulation from representation and facilitating general purpose interaction techniques[7].

The addition of constraints to an interactive graphical application creates a variety of issues which system builders must be concerned with. Many of these issues stem from interface concerns and the need to handle non-linear relationships. Achieving the needed performance and dynamicness from the non-linear solvers requires careful attention. However, support for these can be provided in a general purpose manner.

References

[1] Sherman R. Alpert. Graceful interaction with graphical constraints. *IEEE Computer Graphics and Applications*, pages 82–91, March 1993.

[2] Eric Bier and Maureen Stone. Snap-dragging. *Computer Graphics*, 20(4):233–240, 1986. Proceedings SIGGRAPH '86.

[3] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.

[4] Phillip Gill, Walter Murray, and Margret Wright. *Practical Optimization*. Academic Press, New York, NY, 1981.

[5] Michael Gleicher. Briar - a constraint-based drawing program. In *SIGGRAPH Video Review*, volume 77, 1992. CHI '92 Formal Video Program.

[6] Michael Gleicher. Integrating constraints and direct manipulation. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, pages 171–174, March 1992.

[7] Michael Gleicher. Building interactive systems with differential constraints. submitted for publication, February 1993.

[8] Michael Gleicher and Andrew Witkin. Differential manipulation. *Graphics Interface*, pages 61–67, June 1991.

[9] Michael Gleicher and Andrew Witkin. Snap together mathematics. In Edwin Blake and Peter Weisskirchen, editors, *Advances in Object Oriented Graphics 1: Proceedings of the 1990 Eurographics Workshop on Object Oriented Graphics*. Springer Verlag, 1991. Also appears as CMU School of Computer Science Technical Report CMU-CS-90-164.

[10] Michael Gleicher and Andrew Witkin. Drawing with constraints. submitted for publication, October 1992.

[11] Michael Gleicher and Andrew Witkin. Through-the-lens camera control. *Computer Graphics*, 26(2):331–340, July 1992. Proceedings Siggraph '92.

[12] Michael Gleicher and Andrew Witkin. Supporting numerical computations in interactive contexts. In Tom Calvert, editor, *Proceedings Graphics Interface*, May 1993. To Appear.

[13] Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic, 1989.

[14] Ralph D. Hill. A 2-d graphics system for multi-user interactive graphics based on objects and constraints. In E. Blake and P. Weisskirchen, editors, *Advances in Object Oriented Gr 1: Proceedings of the 1990 Eurographics Workshop on Object Oriented Graphics*, pages 67–92. ger Verlag, 1991.

[15] Masao Iri. History of automatic differentiation and rounding error estimation. In Andreas Griewank and George Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 3–16. SIAM, January 1991.

[16] David Kurlander and Stephen Feiner. Inferring constraints from multiple snapshots. Technical Report CUCS-008-91, Columbia University, May 1991.

[17] John Harold Maloney. *Using Constraints for User Interface Construction.* PhD thesis, University of Washington, 1991. Appears as Computer Science Technical Report 91-08-12.

[18] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, and Phillipe Marchal. Comprehensive support for graphical, highly-interactive user interfaces: The garnet user interface development environment. *IEEE Computer*, November 1990.

[19] Yoshiko Nakamura. *Advanced Robotics: Redundancy and Optimization.* Addison-Wesley, 1991.

[20] John Platt. *Constraint Methods for Neural Networks and Computer Graphics.* PhD thesis, California Institute of Technology, 1989.

[21] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, England, 1986.

[22] Steven Sistare. Interaction techniques in constraint-based geometric modeling. In *Proceedings Graphics Interface '91*, pages 85–92, June 1991.

[23] Mark Surles. Interactive modeling enhanced with constraints and physics – with applications in molecular modeling. In *Proceedings of the 1992 Symposium on Interactive Computer Graphics*, pages 175–182, March 1992.

[24] Mark C. Surles. An algorithm for linear complexity for interactive, physically-based modelling of large proteins. *Computer Graphics*, 26(2):221–230, 1992. Proceedings SIGGRAPH '92.

[25] Ivan Sutherland. *Sketchpad: A Man Machine Graphical Communication System.* PhD thesis, Massachusetts Institute of Technology, January 1963.

[26] Charles W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares method. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):93–101, January 1986.

Concurrent Constraint Programming at SICS with the Andorra Kernel Language (Extended Abstract)

Seif Haridi Sverker Janson
Johan Montelius Torkel Franzén Per Brand
Kent Boortz Björn Danielsson Björn Carlson
Torbjörn Keisu Dan Sahlin Thomas Sjöland

SICS, Box 1263, S-164 28 KISTA
Tel +46-8-752 15 00, Fax +46-8-751 72 30
E-mail {seif, sverker}@sics.se

Abstract

SICS is investigating a new generation of languages for symbolic processing that are based on the paradigm of concurrent constraint programming. A wide range of pertinent topics are being studied. In particular our efforts are devoted to producing a high quality programming environment based on the Andorra Kernel Language (AKL), a general purpose concurrent constraint language.

1 Introduction

Concurrent constraint programming (CCP) is a powerful paradigm for programming with constraints while being based on simple concepts [9, 11]. A set (or conjunction) of constraints, regarded as formulas in first-order logic, forms a *constraint store*. A number of *agents* interact with the store using the two operations *tell*, which adds a constraint to the store, and *ask*, which tests if the store either entails or disentails the asked constraint, otherwise waiting until it does. Telling and asking correspond to sending and receiving "messages", thereby providing the basic means for communication and synchronisation for concurrent programming.

The Andorra Kernel Language (AKL) is a concurrent constraint programming language which generalises the above functionality using a small set of powerful combinators [5]. The basic paradigm is still that of agents communicating over a constraint store, but the combinators make possible also other readings, depending on the context, where agents compute functions or relations, serve as user-defined constraints, or as objects in object-oriented programs. A major point of AKL is that its paradigms can be combined. For example, it is quite natural to have a reactive process- or object-oriented top-level in a program, with other components performing constraint solving using don't know nondeterminism. The nondeterminism can be encapsulated, so that it does not affect the process component.

Nondeterminism in AKL is controlled using *stability*, a generalisation of the Andorra principle, which has proven its usefulness in the context of constraint programming (see e.g., [4]).

AKL offers a large potential for parallel execution, no less than that of logic programming languages and dataflow languages [10]. Arbitrary parallel algorithms can be programmed in AKL, which is exemplified by its ability to simulate a PRAM. This ability is given by ports, an extension of the language for process communication, but in the spirit of concurrent constraint programming [6]. Pure functional and logic languages do not have this ability.

SICS is currently developing a programming environment for AKL [7]. Among our goals are to: (1) develop the necessary implementation technology for efficient (sequential and parallel) execution, (2) develop an

execution model which allows different constraint systems to be easily incorporated, (3) offer good interoperability with conventional languages such as C, and (4) investigate large scale applications where combinations of paradigms naturally occur.

Our efforts are divided into the following main lines of activities.

Language Design A language meeting our requirements has been designed: AKL. Possible extensions and generalisations are being investigated. AKL features

- Combinators for (parallel) composition, hiding, conditional choice, nondeterminate choice, committed choice, and solution aggregation
- Improved control and synchronisation, e.g., encapsulation of nondeterminism controlled by stability
- Subsumption of Prolog, CLP, and committed-choice languages such as GHC, Parlog, and Strand
- Support for concurrent object-oriented programming
- Support for arbitrary parallel algorithms

Current and future work includes investigation of maximizing (minimising) combinators, and *engines*, which provide an AKL computation with the ability to inspect and control another computation.

Implementation A prototype implementation of full AKL has been developed. Experiments indicate that performance, using an optimising compiler being developed, will be no less than that of state of the art implementations of Prolog (e.g., SICStus Prolog). The implementation consists of the following main parts.

- A compiler (written in AKL) to an abstract machine
- A threaded emulator (written in C)
- A Prolog-style debugger

Current and future work includes developing an optimising compiler and a parallel implementation.

Constraint Systems A number of different constraint systems are being considered. The prototype implementation is parameterised with constraint systems, which may be added as separate modules. The following constraint systems are addressed. (See also the following section.)

- Herbrand and rational trees serve as a foundation.
- Feature trees have been implemented.
- Finite domains are being investigated.
- Complete Herbrand (closed under logical combinators) is being designed.

Current work also includes the definition of a generic constraint interface.

Formal Aspects Soundness and completeness results for a logical interpretation of AKL have been shown and a proof system for programs is being investigated.

Analysis and Transformation Program analysis by abstract interpretation and program transformation by partial evaluation are studied, both with the aim to improve performance.

In the remainder of this paper a short section will first summarise the constraint systems that have been considered, and the rest is devoted to an explanation of AKL, since it is assumed that this language is largely unknown to the present readership.

2 Constraint Systems

A *constraint system* for AKL may in principle be any first-order theory, closed under conjunction and existential quantification, for which decision procedures for satsifiability and entailment can be provided. Of course, a large body of work on practically motivated constraint systems exists. We have so far considered Herbrand and rational trees, feature trees, finite domains, and complete Herbrand. This investigation will surely be extended to other systems in the future.

2.1 Herbrand and Rational Trees

Herbrand is a theory of equality of terms (the Clark equality theory). The difference between Herbrand and rational trees is that the latter provides solutions to constraints of the form 'X = f(X)'. Traditionally, logic programming is based on Herbrand or rational trees.

For example 'T = tree(1,L,R)' and 'L = [1,2,3|S]', as may be found in Prolog programs, are Herbrand constraints.

2.2 Feature Trees

Feature trees are formed from constraints of the form 'X f Y', where 'f' is a *feature*, a feature being anything which may serve as a label [1]. Intuitively, the feature constraint associates with X a "property" f having the value Y, thus regarding X as a map from properties to values.

2.3 Finite Domains

Finite domain constraints encodes efficiently properties for finite sets [12, 13].

For example, 'X in 1..5' restricts X to be in the range 1 to 5, 'Z in 4..8' correspondingly for Z, and by adding 'X > Z', we may conclude 'X = 5' and 'Z = 4'.

2.4 Complete Herbrand

Complete Herbrand is Herbrand closed under the usual (first-order) logical combinators [8]. This makes the decision procedures for satisfiability and entailment considerably more complex, while offering, among other things, the potential for a more powerful treatment of negation.

3 Language Design

In this section AKL is introduced step by step, introducing one language construct at a time, also explaining its behaviour. There is no space for a formal definition of the computation model (even though it is fairly concise). An explanation of *solution aggregates* has also been omitted for reasons of space.

For a formal computation model see [5], and [3] which also defines the logical interpretation. Observe that these are based on a clausal syntax which is equivalent to the current combinator syntax. Both are available in the AKL programming system.

3.1 Basic Constructs

The agents of concurrent constraint programming correspond to statements being executed concurrently.

Constraints, as discussed in the previous section, appear as atomic statements known as *constraint atoms* (or just *constraints*).

A *program atom* of the form

⟨name⟩(X_1, ..., X_n)

is a defined agent. For example,

plus(X, Y, Z)

is a (plus/3) atom.

The behaviour of atoms is given by *(agent) definitions* of the form

⟨name⟩(X_1, ..., X_n) := ⟨statement⟩.

The variables X_1, ..., X_n must be different. During execution, any atom matching the left hand side will be replaced by the statement on the right hand side. For example,

plus(X, Y, Z) := Z = X + Y.

is a definition (of plus/3).

A *composition statement* of the form

⟨statement⟩, ..., ⟨statement⟩

builds a composite agent from a number of agents. Its behaviour is to replace itself with the concurrently executing agents corresponding to its components.

A *hiding statement* of the form

X_1, ..., X_n : ⟨statement⟩

introduces variables with local scope. The behaviour of a hiding statement is to replace itself with its component statement, in which the variables X_1, ..., X_n have been replaced by new variables.

The *conditional choice statement*

(⟨statement⟩ → ⟨statement⟩
; ...
; ⟨statement⟩ → ⟨statement⟩)

is used to express conditional execution. Its components are called *(guarded) clauses* and the components of a clause *guard* and *body*. A clause may be enclosed in hiding.

The behaviour of a conditional choice statement is as follows. Its guards are executed with corresponding local constraint stores. If the union of a local store with the external stores is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, the choice statement fails. If the first (remaining) guard is successfully reduced to a store which is entailed by the union of external stores, the conditional choice statement is replaced with the composition of the constraints with the body of the corresponding clause.

If a variable Y is hidden in a clause, then, when testing for entailment, the local constraint store is preceded by the expression 'for some Y' (or logically, '∃Y'). For example, in

X = f(a), (Y : X = f(Y) → q(Y))

the asked constraint is '∃Y (X = f(Y))' ('for some Y, X = f(Y)'), which is entailed, since there exists a Y (namely 'a') such that X = f(Y) is entailed.

It is now time for a first small example, illustrating the nature of concurrent computation in AKL. The following definitions will create a list of numbers, and add together a list of numbers, respectively.

```
list(N, L) :=
      ( N = 0 → L = []
      ; L1 : N > 0 → L = [N|L1], list(N - 1, L1) ).
sum(L, N) :=
      ( L = [] → N = 0
      ; M, L1, N1 : L = [M|L1] → sum(L1, N1), N = N1 + M ).
```

The following computation is possible. In the examples, computations will be shown by performing rewriting steps on the state (or configuration) at hand, unfolding definitions and substituting values for variables, etc., where appropriate, which should be intuitive. In this example we avoid details by showing only the relevant atoms and the collection of constraints on the output variable N. Intermediate computation steps are skipped. Thus,

list(3, L), sum(L, N)

is rewritten to

list(2, L1), sum([3|L1], N)

by unfolding the list atom, executing the choice statement, and substituting values for variables according to equality constraints. This result may in its turn be rewritten to

list(1, L2), sum([2|L2], N1), N = 3 + N1

by similar manipulations of the list and sum atoms. Further possible states are

list(0, L3), sum([1|L3], N2), N = 5 + N2
sum([], N3), N = 6 + N3
N = 6

with final state N = 6.

The list/2 agent produces a list, and the sum/2 agent is there to consume its parts as soon as they are created. If the tail of the list being consumed by the sum/2 call is unconstrained, the sum/2 agent will wait for it to be produced (in this case by the list/2 agent).

The simple set of constructs introduced so far is a fairly complete programming language, quite comparable in expressive power to, e.g., functional programming languages.

In the following sections, we will introduce constructs that address the specific needs of important programming paradigms, such as processes and process communication, object-oriented programming, relational programming, and constraint satisfaction. In particular, we will need the ability to choose between alternative computations in a manner more flexible than that provided by conditional choice.

3.2 Don't Know Nondeterminism

Many problems, especially frequent in the field of Artifical Intelligence, and also found elsewhere, e.g., in operations research, are currently solvable only by resorting to some form of search. Many of these admit very concise solutions if the programming language abstracts away the details of search by providing don't know nondeterminism.

For this, AKL provides the *nondeterminate choice* statement.

```
( ⟨statement⟩ ? ⟨statement⟩
; ...
; ⟨statement⟩ ? ⟨statement⟩ )
```

The symbol '?' is read *wait*. The statement is otherwise like the conditional choice statement.

The behaviour of a nondeterminate choice statement is as follows. Its guards are executed with corresponding local constraint stores. If the union of a local store with the external stores is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, the choice statement fails. If only one clause remains, and its guard is successfully reduced to a store which is consistent with the union of external stores, the choice statement is said to be *determinate*. Then, the nondeterminate choice statement is replaced with the composition of the constraints with the body of the corresponding clause. Otherwise, if there is more than one clause left, the choice statement is said to be *nondeterminate*, and it will wait. Subsequent telling of other agents may make it determinate. If eventually a state is reached in which no other computation step is possible, each of the remaining clauses may be tried in different copies of the state. The alternative computation paths are explored concurrently.

Let us first consider a very simple example, an agent that accepts either of the constants a or b, and then does nothing.

```
p(X) :=
      ( X = a ? true
      ; X = b ? true ).
```

The interesting thing happens when the agent p is called with an unconstrained variable as an argument. That is, we expect it to produce output. Let us call p together with an agent q examining the output of p.

```
q(X, Y) :=
      ( X = a → Y = 1
      ; true → Y = 0 ).
```

Then the following is one possible computation starting from

```
p(X), q(X, Y)
```

First p and q are both unfolded.

```
( X = a ? true ; X = b ? true ),
( X = a → Y = 1 ; true → Y = 0 )
```

At this point in the computation, the nondeterminate choice statement is nondeterminate, and the conditional choice statement cannot establish the truth or falsity of its condition. The computation can now only proceed by trying the clauses of the nondeterminate choice in different copies of the computation state. Thus.

```
X = a, ( X = a → Y = 1 ; true → Y = 0 )
Y = 1
```

and

```
X = b, ( X = a → Y = 1 ; true → Y = 0 )
Y = 0
```

are the two possible computations. Observe that the nondeterminate alternatives are ordered in the order of the clauses in the nondeterminate choice statement.

The constructs introduced so far give us (constraint) logic programming in addition to functional programming. Nondeterminism is introduced only lazily. Propagation of known constraints is always given priority. This simple functionality gives us means to solve many constraint satisfaction problems efficiently [4].

Up to this point, the constructs introduced belong to the strictly logical subset of AKL, which has a straightforward interpretation in first-order logic both in terms of success and failure.

3.3 Don't Care Nondeterminism

In concurrent programming, processes should be able to react to incoming communication from different sources. In constraint programming, constraint propagating agents should be able to react to different conditions. Both of these cases can be expressed as a number of possibly non-exclusive conditions with corresponding branches. If one condition is satisfied, its branch is chosen.

For this, AKL provides the *committed choice* statement

```
( ⟨statement⟩ | ⟨statement⟩
; ...
; ⟨statement⟩ | astatementn )
```

The symbol '|' is read commit. The statement is otherwise like the conditional choice statement.

The behaviour of a committed choice statement is as follows. Its guards are executed with corresponding local constraint stores. If the union of a local store with the external stores is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, the choice statement fails. If any of the (remaining) guards is successfully reduced to a store which is entailed by the union of external stores, the committed choice statement is replaced with the composition of the constraints with the body of the corresponding clause.

List merging may now be expressed as follows, as an example of an agent receiving input from two different sources.

```
merge(X, Y, Z) :=
      ( X = [] | Z = Y
      ; Y = [] | Z = X
      ; E, X1, Z1 : X = [E|X1] | Z = [E|Z1], merge(X1, Y, Z1)
      ; E, Y1, Z1 : Y = [E|Y1] | Z = [E|Z1], merge(X, Y1, Z1) ).
```

A merge agent can react as soon as either X or Y is given a value. In the last two guarded statements, hiding introduces variables that are used for "matching" in the guard, as discussed above. These variables are constrained to be equal to the corresponding list components.

3.4 Encapsulated Computations

To avoid unwanted interactions between don't know nondeterministic and process-oriented parts of a program, the nondeterministic part can be encapsulated in a statement that hides nondeterminism. Nondeterminism is encapsulated in the guard of a conditional or committed choice and in the solution aggregation constructs provided by AKL.

The scope of don't know nondeterminism in a guard is limited to its corresponding clause. New alternative computations for a guard will be introduced as new alternative clauses. This will be illustrated using the following simple nondeterminate agent.

```
or(X, Y) :=
      ( X = 1 ? true
      ; Y = 1 ? true ).
```

Let us start with the statement

```
( or(X, Y) | q )
```

The or atom is unfolded, giving

```
( ( X = 1 ? true ; Y = 1 ? true ) | q )
```

Since no other step is possible, we may try the alternatives of the nondeterminate choice in different copies of the closest enclosing clause, which is duplicated as follows.

```
( X = 1 | q
; Y = 1 | q )
```

Other choice statements are handled analogously.

Before leaving the subject of don't know nondeterminism in guards, it should be clarified exactly when alternatives may be tried. A (possibly local) state with agents and their store is (locally) stable if no computation step other than splitting a nondeterminate choice is possible, and no such computation step can be made possible by adding constraints to external constraint stores (if any). Splitting may then be applied to the leftmost possible nondeterminate choice in a stable state.

3.5 Ports for Concurrent Objects

The combinators mentioned above are adequate to model concurrent objects in a style known from concurrent logic programming. A standard example of an object definition is a bank account providing services such as withdrawals, deposits, etc.

```
make_bank_account(S) := bank_account(S,0).

bank_account(Ms, State) :=
    ( Ms = [] → true
    ; A,R : Ms = [withdraw(A)|R] →
          bank_account(R, A-State)
    ; A,R : Ms = [deposit(A)|R] →
          bank_account(R, A+State)
    ; A,R : Ms = [balance(A)|R] →
          A = State, bank_account(R, State)).
```

However it known from experience that communication between objects using streams and merger agents is very awkward.

In AKL we use another communication medium between objects, called *ports*. A port is a binary constraint on a bag (a multiset) of messages and a corresponding stream of these messages. It simply states that they contain the same messages in any order. A bag connected to a stream by a port is usually identified with the port, and is referred to as a port. The open_port(P,S) operation relates a bag P to a stream S, and connects them through a port.

The stream S will typically be connected to an object of the above form. Instead of using the stream to access the object, we will send messages by adding them to the port. The constraint send(P,M) sends the message M to the port P. To satisfy the port constraint, a message sent to a port will immediately be added to its associated stream, first come first served. In this sense the port constraints have the same status as the committed choice statement by committing to a single arbitrary order of messages in the stream associated with the port constraint.

When a port is no longer referenced from other parts of the computation state, when it becomes garbage, it is assumed that it contains no more messages, and its associated stream is automatically closed. When the stream is closed, any object consuming it is thereby notified that there are no more clients requesting its services.

A simple example follows.

```
open_port(P,S), send(P,a),send(P,b)
```

yields

```
P = <a port>, S=[a,b]
```

Ports solve a number of problems that are implicit in the use of stream. Here is a summary, for a detailed description of these see [6].

- Several clients can access the same objects without the need to explicitly merge messages into a single stream.
- Objects can be embedded freely in other data structures provided by AKL.
- Message sending conceptually takes constant time in the computational model of AKL.
- Automatically closing the stream associated to a port, when the port is no longer accessible, provides a good method for garbage collecting concurrent objects.

3.6 Ports for State and Parallel Algorithms

Ports provide AKL with means to incorporate various types of mutable data structures, such as arrays and hashtables. These structures are modelled in AKL as objects connected to a port, but may be implemented very efficiently at a lower level.

For example, a memory cell can be modelled in AKL as an object connected to a port that accepts the messages read(V), write(V), and exchange(V1,V2), to read a value V, to write a value V, and to atomically exchange the current value V1 with V2, respectivley.

This ability allows us to model a parallel random access memory, or parallel access to hash tables in the language, and as a consequence allows us to write many parallel algorithms that cannot be efficiently expressed in pure functional and concurrent logic languages. The following is a typical example.

First we define a shared memory as follows.

```
memory(M) :=
        M = m(C1, ..., Cn),
        cell(C1), ..., cell(Cn).
```

where cell/1 agents are specified as above. M becomes a tuple of ports to cells.

The problem is to, given a binary tree in which the leaves contain numbers in the range $1, \ldots, n$, count the occurrences of each number by a parallel algorithm (as parallel "as possible"). In our solution the occurrences are collected in a table of counters, with indices in the given range. Assume that the memory agent defined above defines a memory of this size. The program traverses the tree, incrementing the counter corresponding to each number found. To guarantee that all nodes have been counted, the program performs the computation in a guard, making the table visible to other agents only when the computation has completed.

```
histogram(T, M) :=
        memory(M), count(T, M) ? true.

count(Tree, Table) :=
        (I,C,K : Tree = leaf(I) →
                arg(I, M, C), send(exchange(K,K+1),C)
        ; L,R : Tree = node(L,R) →
                count(L,M),count(R,M)).
```

This example is due to [2].

4 Concluding Remarks

Current and planned topics at SICS include efficient sequential and parallel implementations parametrised with user-definable constraint systems (in C), implementations of various constraint systems, extensions of

the basic framework, such as engines for meta-level programming, program analysis and program transformation, inter-operability with conventional languages and operating systems, and investigation of formal properties.

An experimental AKL programming system is available from SICS for research purposes. The system consists of a compiler (in AKL) from AKL to an abstract machine, an emulator written in C (including a copying garbage collector), and a Prolog style debugger.

References

[1] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*. ICOT, 1992.

[2] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: extending a parallel, non-strict, functional language with state. In *Functional Programming and Computer Architecture '91*, 1991.

[3] Torkel Franzén. Logical aspects of the Andorra Kernel Language. SICS Research Report R91:12, Swedish Institute of Computer Science, October 1991.

[4] Steve Gregory and Rong Yang. Parallel constraint solving in Andorra-I. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*. ICOT, 1992.

[5] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*. MIT Press, 1991.

[6] Sverker Janson, Seif Haridi, and Johan Montelius. *Research Directions in Concurrent Object-Oriented Programming*, chapter Ports for Objects in Concurrent Logic Programs. MIT Press, 1993. To appear.

[7] Sverker Janson and Johan Montelius. The design of the AKL/PS 0.0 prototype implementation of the Andorra Kernel Language. ESPRIT deliverable, EP 2471 (PEPMA), Swedish Institute of Computer Science, 1992.

[8] Torbjörn Keisu. Hcl. SICS research report, Swedish Institute of Computer Science, 1993. Fortcoming.

[9] Michael J. Maher. Logic semantics for a class of committed choice programs. In *Logic Programming: Proceedings of the Fourth International Conference*. MIT Press, 1987.

[10] Remco Moolenaar and Bart Demoen. A parallel implementation of AKL. CW-report, Department of Computer Science, Katholieke Universiteit Leuven, 1991.

[11] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1990. To be published by MIT Press.

[12] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[13] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Technical report, Computer Science Department, Brown University, 1991.

Fourier's Elimination: Which to Choose ?

Jean-Louis Imbert[1]

G.I.A. Parc Scientifique et Technologique de Luminy
163, Avenue de Luminy, case 901
F-13288 Marseille cedex 9 (France)
Email: imbert@gia.univ-mrs.fr

Abstract

Variable elimination is of major interest for Constraint Logic Programming Languages [JaLa86], and Constraint Query Languages [KKR90], where we would like to eliminate auxiliary variables introduced during the execution of a program. This elimination is always suitable for final results. It can also increase the efficiency of the intermediary processes. We focus on linear inequalities of the form $ax \leq b$, where a denotes a n-real vector, x an n-vector of variables, b a real number, and the juxtaposition ax denotes the inner product. In this paper, we will focus exclusively on methods related to Fourier's elimination [Fourie]. Our aim is to make visible the links between the different contributions of S.N. Černikov [Cern63], D.A. Kolher [Kohl67], R.J. Duffin [Duff74], JL.J. Imbert [Imbe90], and J.Jaffar, M.J. Maher, P.J. Stuckey and R.H.C. Yap [JMSY92]. This study, which has never been done before, is of great interest for languages such as CHIP, CLP($\Re$) and Prolog III. We show that the three methods proposed by Černikov, Kolher and Imbert produce exactly the same output (without more or less redundant inequations), up to multiplying by a non-zero positive scalar. We present and discuss the improvements of Černikov, Duffin, Imbert and Jaffar and al, and propose a new improvement. We give a short analysis of the complexity of the main improvements and discuss the choice of the method in relation to the problem at hand. We propose a pattern algorithm. Finally, we conclude with a comparative assessment through a brief example and a few remarks.

Keywords: Variable Elimination, Quantifier Elimination, Projection, Constraint Logic Programming, Output.

1 Introduction

Variable elimination is of major interest for Constraint Logic Programming Languages [JaLa86], and Constraint Query Languages [KKR90], where we would like to eliminate auxiliary variables introduced during the execution of a program. This elimination is always suitable for final results. It can also increase the efficiency of the intermediary processes. We focus on linear inequalities of the form $ax \leq b$, where a denotes a n-real vector, x an n-vector of variables, b a real number, and the juxtaposition ax denotes the inner product. This type of constraint occurs in CLP languages such as CHIP [DVSA88, VHen89], CLP($\Re$) [JaMi87], and Prolog III [Colm87, Colm90]. A constraint system is a conjunction of constraints. Using matrix notation, an inequation system $\{a_i x \leq b_i \mid i = 1, \ldots, m\}$ can be written $Ax \leq b$, where A denotes an $(m \times n)$-matrix, and b an m-real vector. The main problem we face during the variable elimination process in linear inequation systems, is the size of the output. It is doubly exponential. Variable elimination in inequation systems has been extensively investigated. Among these investigations one can cite the C. and JL. Lassez method [LaLa91], which globally eliminates in one single operation the set of unwanted variables. It is based on

[1]Supported by ACCLAIM Project.

semantic properties of projection and of convex hull. It makes it possible to obtain approximations. Nevertheless, so far, mainly methods derived from Fourier's elimination [Fourie] are used in CLP languages. For example, the programming language CLP($\Re$) outputs its results after eliminating undesirable variables using a method related to Fourier [JMSY92]. This kind of method carries out an incremental elimination of variables, one after another. The major problem comes from the size of intermediary systems.

In this paper, our interest will be exclusively focused on methods derived from Fourier's elimination. Among the improvements in these eliminations are the contributions of S.N. Černikov [Cern63], D.A. Kolher [Kohl67], R.J. Duffin [Duff74], J.L.J. Imbert [Imbe90], and J.Jaffar, M.J. Maher, P.J. Stuckey and R.H.C. Yap [JMSY92]. There are basically three approaches represented in these methods: the general algebraic approach of Černikov, the matricial algebraic approach of Kohler, and the graph, or more specifically, the tree approach of [Imbe90]. These trees reflect the way in which the new inequations are constructed as the elimination proceeds. This last approach shows clearly that the methods proposed by Černikov, Kohler and Imbert are equivalent. This is far from being in evidence as Kohler remarks (see Section 4).

The aim of this paper is to make visible the links between these different above-mentioned contributions. This study, which has not been done before, is of great interest. We show that the three methods proposed by Černikov, Kolher and Imbert produce exactly the same output (without more or less redundant inequations), up to multiplying by a non-zero positive scalar. We present and discuss improvements by Černikov, Duffin, Imbert and Jaffar and al, and propose a new improvement. We give a short analysis of the complexity of the main improvements and discuss the choice of the method depending on the problem at hand. We propose a pattern algorithm. Finally, we conclude with a comparative assessment using a brief example and a few remarks.

The rest of this paper is organized in the following way: Section 2 presents basic concepts necessary for understanding the other Sections. In Section 3, we introduce the parts of the initial system (called minimal parts) which can produce a relevant final inequation, and present the Černikov-Fourier algorithm [Cern63]. In Section 4, we introduce the characterization of minimal parts and give the Kohler-Fourier algorithm [Kohl67]. Moreover, in this Section, we show that the algorithms of Černikov-Fourier and Kohler-Fourier produce exactly the same final system (without more or less redundant inequations), up to multiplying each inequation by a non-zero positive scalar. Section 5, successively presents improvements by Černikov, Duffin, Imbert and Jaffar and all, and introduces some new precisions about redundancy. Furthermore, we show (subsection 5.4) a new improvement of the Černikov minimal part method, dividing in half the average number of comparisons. In Section 6 we discuss the complexity of the various improvement contributions and the choice between the minimal part method (Černikov) and the matricial method (Kohler). Then, we give a pattern variable elimination algorithm for inequation systems. Section 7 compares through a brief example, the contributions of Černikov, Kohler and Imbert [Imbe90]. Finally, in Section 8, we conclude with a few remarks.

2 Preliminaries

Let x denote the vector $(x_1, \ldots, x_n)$. Let x' be the vector of variables to be retained, and let x'' be the vector of variables to be eliminated. We will abuse the language by writing $x = (x', x'')$. In the same way, $a = (a_1, \ldots, a_n)$ denotes a real or rational vector, and a will be written (a', a''), respective of the subscripts of x' and x''. The inequation system $Ax \leq b$ can then be written $A'x' + A''x'' \leq b$. To eliminate the variables of x'', the problem at hand is to find a system $Cx' \leq d$ as concise as possible for which, if (a', a'') is in the solution set of $A'x' + A''x'' \leq b$, then a' is in the solution set of $Cx' \leq d$, and reciprocally, if a' is a solution of system $Cx' \leq d$, then there is a'' such that (a', a'') is a solution of system $A'x' + A''x'' \leq b$. We will say that $Cx' \leq d$ and $A'x' + A''x'' \leq b$ are *equivalent on* x'. It can be shown that each inequation of the final system $Cx' \leq d$ is a linear combinatory with positive coefficients of inequations of initial system (this is a consequence of Fourier's elimination).

2.1 Fourier's Algorithm

Let $a_1 x \leq b_1$ and $a_2 x \leq b_2$ be two inequations, and let $a_{1,1}$ and $a_{2,1}$ be the coefficients of the variable x_1 respectively in the first and second inequations. Let us assume that $a_{1,1} > 0$ and $a_{2,1} < 0$. Then,

$$-a_{2,1}(a_1 x) + a_{1,1}(a_2 x) \leq -a_{2,1} b_1 + a_{1,1} b_2 \quad (1)$$

is a consequence of the two initial inequations, and x_1 does not occur in it. Let $Ax \leq b$ be an inequation system, and V the variables which occur in it. Let $\widetilde{A}x \leq \widetilde{b}$ be the system obtained from $Ax \leq b$, by removing all inequations which x_1 occurs in, and replacing them with all the inequations of type (1) above, we can construct from any pair of removed inequations. It can be proved that $\widetilde{A}x \leq \widetilde{b}$ and $Ax \leq b$ are equivalent on $V - \{x_1\}$. This operation must be successively repeated for each variable to be eliminated.

2.2 Redundancy and size of the intermediary systems

A drawback of Fourier's elimination is the production of redundancies which overwhelms the system. It is here that the improvements of S.N. Černikov [Cern63], D.A. Kolher [Kohl67], R.J. Duffin [Duff74], JL.J. Imbert [Imbe90], and J.Jaffar, M.J. Maher, P.J. Stuckey and R.H.C. Yap [JMSY92], come. One way to mitigate this drawback is to associate each inequation with information to memorize the way in which it has been produced. This information deals with: the inequations of the initial system used to produce this inequation, the variables which are effectively eliminated during pair gathering, and, finally, the other variables eliminated during the previous variable eliminations. Some relations between the quantities of elements occuring in these three kinds of variables, make it possible to detect whether a new inequation is redundant or not.

Another drawback of Fourier's method is the large increase in the number of constraints in the intermediary systems. One of the main reasons is that the historic method quickly detects most of the redundancies, but it does not detect all of them (only those due to the construction, i.e. due to $A''x''$, but not those due to $A'x'$). Besides, redundancies have a tendency to spread as they multiply very quickly. The other reason concerns the nature of Fourier's method: independently of the redundancies, the number of inequations has a tendency to increase, before to decrease when few variables remain to be eliminated. A remedy for the first cause is to use a general method of redundancy removing [Telg81, KLTZ83, LHMA89, ImVH92a]. As for the second cause, which is structural, it cannot be suppressed. Other methods are necessary for remedying this inconvenience. Numerous methods have been proposed, including one already cited and set out in [LaLa91] which is particularly interesting as it creates a minimal representation of the final system and does not use intermediary systems.

2.3 Some results

Let S be the linear inequation system $\{a_1 x \leq b_1, \ldots, a_n x \leq b_n\}$. It is known (Farkas [Shri86, p87-90]), that the inequation $cx \leq d$ is a consequence of S if and only if there exist positive coefficients (≥ 0) $\alpha_0, \alpha_1, \ldots, \alpha_n$, such that $c = \alpha_1 a_1 + \ldots + \alpha_n a_n$ and $d = \alpha_0 + \alpha_1 b_1 + \ldots + \alpha_n b_n$. The inequation $cx \leq d$ is said to be an ***affine combinatory with positive coefficients*** of inequations of S. A ***linear combinatory*** is an affine combinatory such that $\alpha_0 = 0$. Hence, an inequation of S is redundant in S if and only if it is an affine combinatory with positive coefficients of the other inequations of S. It is ***strongly redundant*** if for at least one affine combinatory $\alpha_0 \neq 0$, ***weakly redundant*** in other cases.

Lastly, let us notice that, from Fourier's elimination, each inequation obtained after variable elimination in an inequation system S, is a linear combinatory with positive coefficients of the inequations of S.

3 Minimal subset

As a result of Fourier's elimination, each inequation of the final system $Cx' \leq d$ is a linear combinatory with non-negative coefficients of inequations of the initial system $A'x' + A''x'' \leq b$. Hence, we look at the vectors $w = (w_1, \ldots, w_m)$ such that $wA'' = 0''$. ($0''$ denotes the vector a'' with all its components equal to zero. The vector a'' has been introduced in Section 2. We will abuse the language by writing 0 instead of $0''$ if there is no ambiguity. In the same way, we will write 0 instead of $0'$). A vector w_1 is less than a vector w_2

(written $w_1 \leq w_2$), if each component of w_1 is less than its corresponding component of w_2. Let F be the cone of non-negative solutions $0 \leq w$ of $wA'' = 0$. From the last remark of Section 2.3, it is evident that:

Lemma 1 The system $\{wA'x' \leq wb \mid w \in F\}$ is equivalent to the final system $Cx' \leq d$.

However, this system is unnecessarily large. According to [Cern63], it is sufficient to take a base of F. A *base* of F is an irreducible subset of elements of F which generate F. Indeed, on the one hand, if w_1 and w_2 are two elements of F identical up to multiplying by a non-zero scalar, then the inequations $w_1A'x' \leq w_1b$ and $w_2A'x' \leq w_2b$ are equivalent. On the other hand, from [Cern60, Section 3 p 297-298] it suffices that the number of non-zero components of w is less than the number of non-zero components of x'' plus 1. As a matter of fact, according to [Cern63], it is sufficient to take a base of F. A *base* of F is an irreducible subset of elements of F which generates F. Here, irreductible is for inclusion. Since F is a cone, a vector is *generated* by other vectors if it is a linear combinatory with non-negative coefficients of these other vectors.

Two elements of F are *essentially different* if they do not differ from one another by a positive scalar multiple. A *minimal vector* is a non-zero vector of F such that the only essentially different vector of F less than itself, is the zero-vector. Let G be a base of essentially different minimal vectors of F. Then, using lemma 1, it is evident that:

Theorem 2 The systems $Cx' \leq d$ and $\{wA'x' \leq wb \mid w \in G\}$ are equivalent.

Corollary 3 If w_1 and w_2 are two elements of G, the zero-components of which coincide with each other, then $w_1 = w_2$.

Proof If $w_1 \neq w_2$, there is a linear combinatory of these two vectors which produces a non-zero element of F less than each of them. Indeed, it can be found $k > 0$, such that $kw_1 \leq w_2$. Then, continuously increase k with $kw_1 \leq w_2$, until at least one non-zero component of kw_1 is equal to the corresponding component of w_2. Then, $w_2 - kw_1$ is non-zero positive and is in F. So, w_1 and w_2 are not minimal, and then are not in G. □

Each minimal vector of F or G, is associated with an inequation subset of $A'x' + A''x'' \leq b$ (the lines of the initial subset corresponding to the non-zero components of the minimal vector). This subset is said to be *minimal for* x'', or simply *minimal* if there is no ambiguity.

¿From [Cern60, section 3 p 297-298] and [Cern63], theorem 2 can be used at each step in Fourier's elimination, yielding a correct algorithm. Then, for each inequation i, we memorize the subset H_i of initial inequations (i.e. of initial system $Ax \leq b$) which produced i. H_i is called a *historical subset*. Clearly, for each initial inequation i, $H_i = \{i\}$. In Fourier's algorithm, only inequations with minimal historical subsets are retained. An algorithm using these properties is given in Figure 1.

Notice that the historical subset of the new inequation, and the comparison can be processed before producing the inequation. Hence, the inequation is not created if at least one existing historical subset is included in its historical subset.

4 Characterization of minimal subsets

The main problem we face in the previous method is that the historical subset of each inequation of $Cx' \leq d$ must be compared to the historical subset of every other inequation of $Cx' \leq d$. The following theorem makes it possible to overcome this drawback. In the following, the notion of rank of vector space, affine space, vector system and matrix is assumed known.

Theorem 4 Let $A'x' + A''x'' \leq b$ be an inequation system. Let F be the cone defined in Section 3. An element w of F is minimal iff the sub-matrix A''_w formed from lines of A'' related to non-zero components of w has as its rank, the number of lines of A''_w minus 1.

Proof Since $wA'' = 0$, this rank is at most the number of lines of A''_w minus 1. If it is smaller, there exists any vector v of F with at least one non-zero positive component, such that for each non-zero component of v, the corresponding component of w is non-zero, and such that $vA'' = 0$. Without loss of generality, v

Input: $Ax \leq b$, a system of inequations.
Assume that $x = (x', x'')$, where $x'' = (x_1, \ldots, x_k)$ is the vector of variables to be eliminated.
Output: A system $Cx' \leq d$ equivalent to $Ax \leq b$ on x'

begin

1. At the start, the inequation system S is equal to $Ax \leq b$.
2. **For** x_j being successively the variables $x_1, \ldots, x_k$,
   - 2.1. Remove from S, each inequation with non-zero coefficient of x_j.
   - 2.2. For each pair of removed inequations $a_1 x \leq b_1$ and $a_2 x \leq b_2$, with components $a_{j,1}$ and $a_{j,2}$ of x_j respectively positive and negative, insert the inequation $a_{j,1}a_2 x - a_{j,2}a_1 x \leq a_{j,1}b_2 - a_{j,2}b_1$ in S. If the historical subsets of the inequations of the pair are H_1 and H_2 respectively, then, the historical subset of the new inequation is $H_1 \bigcup H_2$.
   - 2.3. Remove from S each inequation with a historical subset including at least one historical subset of the remaining inequations.
3. Return the system S which is then of the form $Cx' \leq d$.

end

Figure 1: Černikov-Fourier's Algorithm

Input: $Ax \leq b$, an inequation system.
Assume that $x = (x', x'')$, where $x'' = (x_1, \ldots, x_k)$ is the vector of variables to be eliminated.
Output: A system $Cx' \leq d$ equivalent to $Ax \leq b$ on x'

begin

1. we begin with the system of inequations S equal to $Ax \leq b$.
2. **For** x_j being successively the variables $x_1, \ldots, x_k$,
   - 2.1. Remove from S, each inequation with a non-zero coefficient of x_j.
   - 2.2. **For each** pair of removed inequations $a_1 x \leq b_1$ and $a_2 x \leq b_2$, with components $a_{j,1}$ and $a_{j,2}$ of x_j respectively positive and negative.
     - 2.2.1 If the historical subsets of the inequations of the pair are H_1 and H_2 respectively, then, the historical subset of the new inequation is $H_1 \bigcup H_2$.
     - 2.2.2 If the rank of A''_H is its number of lines minus 1, insert in S the new inequation $a_{j,1}a_2 x - a_{j,2}a_1 x \leq a_{j,1}b_2 - a_{j,2}b_1$.
3. Return the system S which is then of the form $Cx' \leq d$.

end

Figure 2: Kohler-Fourier's Algorithm

can be assumed less than w, with equality for at least one non-zero component (if necessary, multiply by an appropriate scalar). As a result, $w - v$ is a non-zero vector of F, then w is not minimal. □

In fact, Černikov defines a minimal vector (which he calls *fundamental element*) as a vector w for which the rank of the matrix A''_w is the number of its lines minus 1 [Cern63, p 1520]. He then says that the maximal systems of essentially different minimal elements of F are identical with its bases. Kohler used this result in Fourier's elimination algorithm. In the algorithm of Figure 2, if H is the historical subset associated with an inequation, A''_H will denote the matrix formed from lines and columns of A'' which respectively correspond to elements of H and to columns of $x_1, \ldots, x_j$.

Note that, contrary to the Černikov-Fourier Algorithm, the detection of minimal subsets is performed at the time of the creation of the new inequations. That is to say, whether a subset is minimal or not is detected at the time of its creation. This is the opposite of Černikov-Fourier Algorithm, in which some

inequations are temporarily kept until a new inequation with a smaller historical subset is created. As a result, time used to create non-retained inequations, and an unnegligible place used for intermediary storage can be saved.

However, it can be asked whether this method detects when the same inequation can be obtained from the same minimal subset in more than one way. Kohler [Kohl67, p 23]: *"I have been unable to prove that we can discount the possibility of the Fourier-Motzkin Method generating more than one extreme vector from the same half-line. Should this occur we need only keep one of them"*. The tree approach of [Imbe90] makes it possible to give a simple answer to this question using its unicity theorem (p 121). This theorem can be translated as follows:

Theorem 5 [Unicity theorem] For each minimal subset, Fourier's algorithm, modified by Černikov or Kohler, can produce only one inequation and only in one way.

Corollary 6 The algorithms Černikov-Fourier and Kohler-Fourier rigorously produce the same final system without more or less redundant inequation.

Proof It is an immediate consequence of the previous theorem. □

Remark : Theorem 5 is valid only if, at each Fourier step, each inequation associated with a non-minimal historical subset is discarded. Otherwise uniqueness is not guaranteed and the Černikov method is preferred.

Another very interesting result shown in [Imbe90] is the independence of the order in which the variables are eliminated:

Theorem 7 Whatever the order in which the variables are eliminated, the Černikov-Fourier algorithm or the Kohler-Fourier algorithm rigorously produce the same final system, without more or less redundant inequation.

5 Quick detection of minimal or non-minimal subsets

The main problem we face in the previous two methods is that the minimal subset detection operation is very costly. In this section, we present various known solutions to this problem and we give some new answers. Most of these improvements can be applied independently to both methods.

5.1 Upper limit of the rank

The first improvement was provided by the precursor of the previous two methods. Though Černikov presents the following improvement in [Cern63], we can find its foundation in [Cern60, p 296 corollary 2].

Theorem 8 After k variable eliminations, if the historical subset associated with an inequation has more than $k+1$ elements, then this historical subset is not minimal.

In this case, the detection cost is very low. Černikov and Kohler included this detection in their algorithms.

5.2 Passive variables

Duffin, in [Duff74, p 90], introduces the active or passive variable concept. A variable is *active* during its elimination if there is at least one pair of inequations in the sense of Fourier's elimination. Otherwise the variable is said to be *passive*. Let x_j be a passive variable. Its elimination can suppress some inequations from the system, but does not add any. This comes from the fact that the coefficients of x_j in the inequations are either all positive ($0 \leq$) or all negative (≤ 0). Then if we delay the elimination of such a variable, since Fourier's elimination uses only linear combinatories with positive coefficents, the coefficients of x_j will all have the same sign. In particular, the inequations with non-zero coefficients of x_j, generate inequations with non-zero coefficients of x_j. These inequations will all be rejected in a subsequent elimination of this variable. Thus,

Theorem 9 After the elimination of k variables, p of which are passive, if in a historical subset more than $k+1-p$ elements occur, then this historical subset is not minimal.

5.3 Upper and lower limits of the rank

So far, the improvements look globally. However, the minimal subset is a local concept in that a minimal part depends only on what is included within it. Hence, it is sufficient to look at the eliminated variables occuring in at least one inequation of the historical subset [Imbe90].

Assume that the variables eliminated from the initial system are $x_1, \ldots, x_k$. We will say that they are ***officially eliminated***, and will write O_k the set of these variables. For each inequation i produced, the set O_k can be divided into three disjointed subsets: the subset of ***effectively eliminated*** variables denoted E_i, the subset of ***implicitly eliminated*** variables denoted I_i, and the other variables. A variable is said to be effectively eliminated for i, if its official elimination produces at least one of its ancestors (initial or intermediate inequations used to produce i). A variable is said to be implicitly eliminated for i, if the following three conditions are satisfied: it occurs in at least one inequation of H_i, it does not occur in i, it is not effectively eliminated for i. (an exemple is given in Section 7). In [Imbe90] the following two theorems are shown, where $Card(S)$ denote! s the number of elements of S:

Theorem 10 [First acceleration theorem] If H_i is a minimal subset, then the following relation is satisfied:

$$1 + Card(E_i) \leq Card(H_i) \leq 1 + Card(E_i \bigcup (I_i \bigcap O_k))$$

In this same paper it is shown that whatever H_i, minimal part or not, the left inequality is satisfied. The right inequality gives an upper limit less than or equal to the one of theorem 8 above. The Duffin improvement is always global and still applicable: we only have to suppress passive variables from O_k. When the first acceleration theorem quickly detects non-minimal parts, the second acceleration theorem quickly detects minimal parts:

Theorem 11 [Second acceleration theorem]
Let i be an inequation such that $1 + Card(E_i) = Card(H_i)$, then H_i is minimal.

This theorem avoids a heavy verification burden. The cost of these two theorems is very low. It linearly depends on the number of eliminated variables. The costly research operation of minimal subsets, either by comparison of minimal parts, or by computing a matrix rank, needs to be done only when

$$1 + Card(E_i) < Card(H_i) \leq 1 + Card(E_i \bigcup (I_i \bigcap O_k)).$$

Thus, if there is no implicitly eliminated variable, these two theorems are sufficient. In return, the algorithm performances decrease when the number of implicitly eliminated variables increases.

5.4 Comparison number

The following improvement deals only with the Černikov-Fourier method. The set G is a base of F, and there is a one-to-one map between G and the set of minimal subsets. Then, for each element j of a non-minimal subset P, there is a minimal part included in P, in which j occurs. As a result, if we take an ordering on the initial inequations, the comparison between subsets with the same first element is sufficient. So, the average number of comparisons is divided in half. However, more storage space is needed for intermediary results.

5.5 Redundancies

Let us consider the initial inequation system $A'x' + A''x'' \leq b$. So far, the only redundancies suppressed during the elimination process of x'', are the ones due to A''. However, if we take into account $A'x'$, other redundancies may appear. I do not know a method which suppress all redundancies, compatible with one of the Fourier's elimination methods presented above. However, in some cases, the coexistence is possible. In [JMSY92], it is shown that strong redundancies[2] produced by Fourier's algorithm with the previous improvements, can be suppressed without subsequent damage.

Theorem 12 Every inequation, at least one ancestor of which is strongly redundant, is strongly redundant.

[2] the supporting hyperplane of which, is far from the solution set of the inequation system.

Proof Let $ux \leq v$ be a strongly redundant inequation. Let us assume that this inequation is equal to the linear combinatory $(\sum_{i=1}^{i=p} \alpha_i a_i)x \leq (\sum_{i=1}^{i=p} \alpha_i b_i) + \alpha_0$. The proof is then trivial since $ux \leq v - \alpha_0$ is a logical consequence of the inequation system and since each produced inequation is a linear combinatory with positive coefficients of the inequations of the system. □

The systematic detection of strong redundancies is very costly. However, there are some cases in which the detection cost is lower. The quasi-redundancy is a special case of strong redundancy. An inequation $ux \leq v$ is quasi-redundant in $Ax \leq b$ if there is another inequation of that system written $ux \leq v - r$ up to multiplied by a positive scalar, where r is a non-zero positive constant [LHMA89]. The quasi-redundancy detection is not excessively expensive, because it is roughly a one-to-one comparison of constraints with each other.

Remark : Assume that all or part of the strong redundancies are suppressed from an intermediary system $\mathcal{C}_i$, and that $\mathcal{C}_i$ is obtained by a derived Fourier's elimination method $\mathcal{M}$. Let $\mathcal{K}_i$ be the subsystem of $\mathcal{C}_i$ so obtained. To take advantage of this, we have to be sure that in the next steps, all the redundancies detected by $\mathcal{M}$ applied on $\mathcal{C}_i$, will not occur in $\mathcal{K}_{i+1}$ when this same method is applied on $\mathcal{K}_i$. If this is the case, we will say that the method $\mathcal{M}$ is *fully compatible* with the partial or full deletion of strong redundancies. Otherwise, we have to find a means to detect all strong redundancies at each step. And this detection is too costly, and then impracticable.

The methods of Černikov-Fourier and Kohler-Fourier, are opposed in that the first detects the minimal subset by comparison with each of the others, whereas the second only needs to know the inequations of its historical subset. As a result, if some minimal subsets are missing due to strong redundancy deletions, the comparison method can be put on the wrong track. On the contrary, this is not the case for the matricial method. In conclusion,

Proposition 13 Any partial deletion of strong redundancies is fully compatible with the Kohler-Fourier elimination method, but is not fully compatible with the Černikov-Fourier elimination method.

Then, the partial deletion of strong redundancies is not suitable for the Černikov-Fourier elimination method.

6 Comparison or matricial computation ?

6.1 Complexity

The incorporation of improvements from theorems 8, 9, 10 and 11, results in a great reduction in the cost of the elimination algorithm. If m_0 is the number of inequations of the initial system, and if k is the number of variables to be eliminated, the cost of theorems 8, 10 and 11 is at most $O(m_0 + k)$ for each produced inequation. Remark that the detection of the minimality of historical subsets can always be done before the creation of the new inequations. Thus, time can be saved in cases of rejection.

The cost of minimal subset detection using comparison is, at most, $O(m_0 m)$ for each produced inequation where m is the maximal number of inequations occurring in the intermediary system during the process of elimination. It must be noted that in the case of Černikov-Fourier method, the comparison must be done in both directions of the inclusion. If we use the improvements of theorems 10 and 11, the comparison must be done only for inequations satisfying theorem 10: in both directions of the inclusion between historical subsets of two new inequations which do not satisfy theorem 11, only in one direction when one of the new inequations satisfies theorem 11, no comparison is needed when the two inequations satisfy this same theorem.

The cost of minimal subset detection using matricial computation is, for each produced inequation, at most $O(k^5)$ in infinite precision, $O(k^3)$ otherwise. Thus, when k is low, it is better to use matricial computation, and to change method during the elimination process when k and m move. Theorem 5 and its corollary allow for this change at every moment.

Furthermore, in the choice of method, we have to take into account the fact that the matricial method allows for an independent process of produced inequations. Conversely, if during the elimination of a variable, the comparison is used, there will be comparisons to do until the end of the elimination of that variable.

Input: $Ax \leq b$, an inequation system.
Let us assume that $x = (x', x'')$, where $x'' = (x_1, \ldots, x_k)$ is the vector of variables to be eliminated.
Output: A system $Cx' \leq d$ equivalent to $Ax \leq b$ on x'

begin

1. We start with
   the inequation system S equal to $Ax \leq b$,
   the set O of officially eliminated variables, empty.
2. **For** x_j being successively the variables $x_1, \ldots, x_k$,
   2.1. Suppress from S all the inequations in which the coefficient of x_j is non-zero.
   2.2. If there is at least one pair of suppressed inequations with opposite sign coefficients of x_j, put x_j into O (* Duffin improvement *), and continue to 2.3, otherwise continue to 2.
   2.3. **For each** pair of suppressed inequations $a_1 x \leq b_1$ et $a_2 x \leq b_2$, of which the coefficients $a_{j,1}$ and $a_{j,2}$ of x_j are respectively positive and negative,
      2.3.1 Assume that the sets associated with these inequations are respectively H_1, E_1, I_1 and H_2, E_2, I_2. Compute $H = H_1 \bigcup H_2$, $E = E_1 \bigcup E_2$ and $I = I_1 \bigcup I_2$.
      2.3.2 If $Card(E \bigcup (I \bigcap O)) < Card(H)$ then continue to 2.3. (*Theorem 10*).
      2.3.3 If $Card(E) = Card(H)$ then go to 2.3.5. (*Theorem 11*).
      2.3.4 Analyze using comparison [3] or matricial computation the set H. If it is a minimal part then go to 2.3.5, else continue to 2.3.
      2.3.5 Suppress some strongly redundant inequations (*Theorem 12*). If the new inequation is strongly redundant then continue to , else continue to 2.2.6.
      2.3.6 Put in S the inequation $a_{j,1}a_2 x - a_{j,2}a_1 x \leq a_{j,1}b_2 - a_{j,2}b_1$.
3. Return the system S which is then of the form $Cx' \leq d$.

end

[3] When the comparison method is chosen, also suppress from the system the inequations of which the historical subsets include the one of the new inequation.

Figure 3: Pattern Modified Fourier's Algorithm

Moreover, the comparison method does not always immediately detect redundancies, as a result, this leads to an additional cost because of the intermediary storage. Consequently, the matricial method allows for a degree of parallelism higher than the comparison method. And then, the matricial method is better suited to an additional redundancy deletion such as quasi-redundancy than the comparison method is. In all cases, these two methods need to be used only when both theorems 10 and 11 fail.

6.2 Modified Fourier's Algorithm

In order to take into account the results of theorems 10 and 11, each inequation i is associated with three sets: the set H_i of initial inequations from which i is produced, the set E_i of its effectively eliminated variables, and the set I_i of its implicitly eliminated variables. When i is an initial inequation, $H_i = \{i\}$, and E_i and I_i are empty. The pattern algorithm is described in Figure 3.

Note that steps 2.3.4 and 2.3.5 can be interchanged. Particularly, if the inequations are ordered, one can quickly see when an inequation is quasi-redundant, and avoid the minimality detection for its historical subset [JMSY92].

If the strong redundancies are suppressed, (step 2.3.5), it is advisable to use the matricial computation at step 2.3.4.

7 An example

In the following example, each inequation is associated with a triplet $(H;\ E;\ I)$. H is the historical subset of the inequation, E the set of its implicitly eliminated variables and I the set of its implicitly eliminated variables. Let x_i, $i = 1, \ldots, 5$ be the variables to be eliminated in the following system:

(1)	$0 \leq -1x_3 - 1x_4 - 1x_5 + 1$	$(1;\ -;\ -)$
(2)	$0 \leq +1x_1 + 2x_4 - 1y_2 + 2$	$(2;\ -;\ -)$
(3)	$0 \leq +1x_2 + 2x_5 - 1y_3 + 2$	$(3;\ -;\ -)$
(4)	$0 \leq -2x_2 - 3x_5 + 1y_3 - 1$	$(4;\ -;\ -)$
(5)	$0 \leq +1x_2$	$(5;\ -;\ -)$
(6)	$0 \leq +1x_3$	$(6;\ -;\ -)$
(7)	$0 \leq -1x_1 - 1x_2 + 2x_3 - 1y_1 + 3$	$(7;\ -;\ -)$
(8)	$0 \leq -1x_1 - 1x_2 - 2x_3 - 2x_4 - 2x_5 + 1y_1 + 1y_2 + 1y_3 - 5$	$(8;\ -;\ -)$
(9)	$0 \leq -1x_2 - 1x_5 + 2y_2$	$(9;\ -;\ -)$

In the elimination of variables x_1, x_2, x_3 and x_4, the retained inequations are all accepted according to theorem 11. The cost is then minimal.

The elimination of x_1 gives $O_1 = \{x_1\}$ all the pairs give a retained inequation. The new system is:

$0 \leq -1x_3 - 1x_4 - 1x_5 + 1$	$(1;\ -;\ -)$
$0 \leq +1x_2 + 2x_5 - 1y_3 + 2$	$(3;\ -;\ -)$
$0 \leq -2x_2 - 3x_5 + 1y_3 - 1$	$(4;\ -;\ -)$
$0 \leq +1x_2$	$(5;\ -;\ -)$
$0 \leq +1x_3$	$(6;\ -;\ -)$
$0 \leq -1x_2 - 1x_5 + 2y_2$	$(9;\ -;\ -)$
$0 \leq -1x_2 + 2x_3 + 2x_4 - 1y_1 - 1y_2 + 5$	$(2.7;\ x_1;\ -)$
$0 \leq -1x_2 - 2x_3 - 2x_5 + 1y_1 + 1y_3 - 3$	$(2.8;\ x_1;\ x_4)$

The elimination of x_2 gives $O_2 = \{x_1, x_2\}$ and the new system is:

$0 \leq -1x_3 - 1x_4 - 1x_5 + 1$	$(1;\ -;\ -)$
$0 \leq +1x_3$	$(6;\ -;\ -)$
$0 \leq +1x_5 - 1y_3 + 3$	$(3.4;\ x_2;\ -)$
$0 \leq +1x_5 + 2y_2 - 1y_3 + 2$	$(3.9;\ x_2;\ -)$
$0 \leq +2x_3 + 2x_4 + 2x_5 - 1y_1 - 1y_2 - 1y_3 + 7$	$(2.3.7;\ x_1.x_2;\ -)$
$0 \leq -2x_3 + 1y_1 - 1$	$(2.3.8;\ x_1.x_2;\ x_4.x_5)$
$0 \leq -3x_5 + 1y_3 - 1$	$(4.5;\ x_2;\ -)$
$0 \leq -1x_5 + 2y_2$	$(5.9;\ x_2;\ -)$
$0 \leq +2x_3 + 2x_4 - 1y_1 - 1y_2 + 5$	$(2.5.7;\ x_1.x_2;\ -)$
$0 \leq -2x_3 - 2x_5 + 1y_1 + 1y_3 - 3$	$(2.5.8;\ x_1.x_2;\ x_4)$

So far, each pair produces a retained inequation. The elimination of x_3 gives $O_3 = \{x_1, x_2, x_3\}$ and proposes nine pairs for the creation of inequations. Among these nine pairs, two are rejected according to theorem 10. The new system is then:

$0 \leq +1x_5 - 1y_3 + 3$	$(3.4;\ x_2;\ -)$
$0 \leq +1x_5 + 2y_2 - 1y_3 + 2$	$(3.9;\ x_2;\ -)$
$0 \leq -3x_5 + 1y_3 - 1$	$(4.5;\ x_2;\ -)$
$0 \leq -1x_5 + 2y_2$	$(5.9;\ x_2;\ -)$
$0 \leq -1x_4 - 1x_5 + 1$	$(1.6;\ x_3;\ -)$
$0 \leq +1y_1 - 1$	$(2.3.6.8;\ x_1.x_2.x_3;\ x_4.x_5)$
$0 \leq -2x_5 + 1y_1 + 1y_3 - 3$	$(2.5.6.8;\ x_1.x_2.x_3;\ x_4)$
$0 \leq -1y_1 - 1y_2 - 1y_3 + 9$	$(1.2.3.7;\ x_1.x_2.x_3;\ x_4.x_5)$
$0 \leq +2x_4 + 2x_5 - 1y_2 - 1y_3 + 6$	$(2.3.7.8;\ x_1.x_2.x_3;\ x_4.x_5)$
$0 \leq -2x_5 - 1y_1 - 1y_2 + 7$	$(1.2.5.7;\ x_1.x_2.x_3;\ x_4)$
$0 \leq +2x_4 - 2x_5 - 1y_2 + 1y_3 + 2$	$(2.5.7.8;\ x_1.x_2.x_3;\ x_4)$

The elimination of x_4 gives $O_4 = \{x_1, x_2, x_3, x_4\}$ and proposes two pairs for the creation of inequations. According to theorem 10 no new inequation is created. The new system is then:

$$
\begin{array}{ll}
0 \leq +1x_5 - 1y_3 + 3 & (3.4;\ x_2;\ -) \\
0 \leq +1x_5 + 2y_2 - 1y_3 + 2 & (3.9;\ x_2;\ -) \\
0 \leq -3x_5 + 1y_3 - 1 & (4.5;\ x_2;\ -) \\
0 \leq -1x_5 + 2y_2 & (5.9;\ x_2;\ -) \\
0 \leq +1y_1 - 1 & (2.3.6.8;\ x_1.x_2.x_3;\ x_4.x_5) \\
0 \leq -2x_5 + 1y_1 + 1y_3 - 3 & (2.5.6.8;\ x_1.x_2.x_3;\ x_4) \\
0 \leq -1y_1 - 1y_2 - 1y_3 + 9 & (1.2.3.7;\ x_1.x_2.x_3;\ x_4.x_5) \\
0 \leq -2x_5 - 1y_1 - 1y_2 + 7 & (1.2.5.7;\ x_1.x_2.x_3;\ x_4)
\end{array}
$$

The elimination of x_5 gives $O_5 = \{x_1, x_2, x_3, x_4, x_5\}$ and proposes eight pairs for the creation of inequations. Two are accepted according to theorem 11. Two are rejected according to theorem 10. And four are rejected using comparison or matricial computation methods. The final system is:

$$
\begin{array}{ll}
0 \leq +1y_1 - 1 & (2.3.6.8;\ x_1.x_2.x_3;\ x_4.x_5) \\
0 \leq -1y_1 - 1y_2 - 1y_3 + 9 & (1.2.3.7;\ x_1.x_2.x_3;\ x_4.x_5) \\
0 \leq -1y_3 + 4 & (3.4.5;\ x_2.x_5;\ -) \\
0 \leq +4y_2 - 1y_3 + 2 & (3.5.9.;\ x_2.x_5;\ -)
\end{array}
$$

The detailed account of realized operations is given in the table below. In Černikov or Kholer methods column, the first number represents the use number of theorem 8, the right number is the use number of comparison or matricial method. In the right column, theorem 8 is replaced by theorem 10 (first number) and theorem 11 (second number). the third number of this column represents the use number of comparison or matricial method.

step	Černikov/Kohler	+ Theorems 10 and 11
1	0 / 2	0 / 2 / 0
2	0 / 8	0 / 8 / 0
3	2 / 7	2 / 7 / 0
4	2 / 0	2 / 0 / 0
5	0 / 8	2 / 2 / 4
total	4 / 25	6 / 19 / 4

It can be seen, in this example, that the proportions of a heavy detection method used, are completely exchanged depending on the use of theorem 10 and 11: (4/29), or theorem 8: (25/29). The advantage supplied by the two acceleration theorems is greater insomuch as the number of implicitly eliminated variables is lower. The extreme case is the one where this number is zero. In this extreme case, only these two theorems are needed for all decisions. This is generally the case when the coefficients of the constraints are randomly generated.

8 Conclusion

After presenting the approaches of Černikov and Kohler for improving the Fourier elimination algorithm, we have shown that these two approaches produce exactly the same final inequation system. The use of theorems 10 and 11 introduced in [Imbe90] considerably decreases the use of costly minimal subset detection methods. We have seen that the matricial minimal subset detection method is better suited to an additional redundancy deletion than the comparison detection method.

In addition, we have tested the heuristic which eliminates, at each Fourier step, the variable with the least production of inequations. The results obtained are clearly not as good as with any other choice, even at random. In some cases, the computation time increases from half a minute to more than an hour, with an overwhelming production. The lower initial production can quickly become disastrous.

Finally, the generalization of the passive variable concept to variables which only produce non-retained constraints, is incorrect, as is shown in the following example. Let the initial system be:

$$
\begin{array}{lll}
(1) & 0 \leq +x+y+z+t & (1;\ -;\ -) \\
(2) & 0 \leq -x-y+z & (2;\ -;\ -) \\
(3) & 0 \leq +x-y-z & (3;\ -;\ -) \\
(4) & 0 \leq -x+y-z & (4;\ -;\ -)
\end{array}
$$

The elimination of x gives

$$
\begin{array}{ll}
0 \leq +2z+t & (1.2;\ x;\ y) \\
0 \leq -2z & (3.4;\ x;\ y) \\
0 \leq +2y+t & (1.4;\ x;\ z) \\
0 \leq -2y & (2.3;\ x;\ z)
\end{array}
$$

The elimination of y gives the system

$$
\begin{array}{ll}
0 \leq +2z+t & (1.2;\ x;\ y) \\
0 \leq -2z & (3.4;\ x;\ y)
\end{array}
$$

and the elimination of z gives the final system

$$0 \leq t \quad (1.2.3.4;\ x.z;\ y).$$

If we had not taken the variable y into account, the inequation $0 \leq t$ would be rejected. Then the final system would be incorrect. Also, this shows the importance of the implicitly eliminated variables.

For new improvements of the Fourier algorithm, it would be interesting to see in which conditions, the weakly redundant inequations (the ones for which the supporting hyperplanes are adjacent to the solution set of the inequation system) can be deleted without putting the previous improvements of the Fourier elimination on the wrong track. However, whatever the future improvements, we cannot prevent the Fourier elimination from producing a great deal of constraints, with a significant increase in the intermediary steps, before this number decreases when few variables remain in the system.

References

[Cern60] S.N. Černikov. "Contraction of Systems of Linear Inequalities". In Soviet mathematics DOKLADY 1, 1960.

[Cern61a] S.N. Černikov. "The Solution of Linear Programming Problem by Elimination of Unknowns". In Soviet mathematics DOKLADY 2, 1961.

[Cern63] S.N. Černikov. "Contraction of Finite Systems of Linear Inequalities". In Soviet mathematics DOKLADY 4, 1963.

[Colm87] A. Colmerauer. "Opening the Prolog III Universe". In *BYTE*, August 1987, p177–182.

[Colm90] A. Colmerauer. "An Introduction to Prolog III". In *Communications of the ACM*, 33, vol 7, July 1990. Also Version française parue dans les comptes rendus des Dixièmes journées Internationales : Les systèmes experts et leurs applications, Avignon, juin 1990.

[Duff74] R.J. Duffin. "On Fourier's Analyse of Linear Inequality Systems". In *Mathematical Programming Study* 1 (1974)71–95. North-Holland Publishing Company.

[DVSA88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier. "The Constraint Logic Programming Language CHIP". In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.

[Fourie] J.B.J Fourier. reported in : "Analyse des travaux de l'Académie Royale des Sciences, pendant l'année 1824, Partie mathématique". *Histoire de l'Académie Royale des Sciences de l'institut de France* 7 1827, xlvii-lv. (Partial English translation in: D.A. Kohler, "Translation of a Report by Fourier on his work on Linear Inequalities", *Opsearch 10* 1973 pages 38–42).

[Imbe89] JL. Imbert. "Simplification des Systèmes de Contraintes Numériques Linéaires". Thèse de Doctorat de l'Université d'Aix-Marseille II, faculté des Sciences de Luminy, Mai 1989.

[Imbe90] JL. Imbert. "About Redundant Inequalities Generated by Fourier's Algorithm". In P. Jorrand, editor, *Proceedings of the Fourth International Conference on Artificial Intelligence*, AIMSA'90, p 117-127.Varna, 1990, Bulgaria. North-Holland. (received the Best Paper Award of the conference).

[ImVH92a] JL. Imbert and P. Van Hentenryck. "A Note on Redundant Linear Constraints". Technical Report CS-92-11, CS Department, Brown University, 1992. 13 pages.

[JaLa86] J.Jaffar and JL. Lassez. "Constraint Logic Programming". Technical Report 86/73. Dept. of computer science. Manash University (June 1986). An abstract appears in *Proceedings of the 14th Principles of Programming Languages*. Munich. 1987. pp 111–119.

[JaMi87] J.Jaffar, S. Michaylov. "Methodology and Implementation of a CLP System". In *Proceedings of the Logic Programming Conference*. Melbourne, 1987. M.I.T. Press.

[JMSY92] J.Jaffar, M.J. Maher, P.J. Stuckey and R.H.C. Yap. "Output in CLP($\Re$)". In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 987–995, June 1992, Tokyo, Japan.

[KKR90] P.C. Kanellakis, G.M. Kuper and P.Z. Revesz. "Constraint Query Languages". in *Proceedings of the ACM Conference on Principles of Database Systems*. Nashville 1990.

[KLTZ83] M.H Karwan, V. Lofti, J. Telgen and S. Zionts. "Redundancy in Mathematical Programming: a State-fo-the-Art survey". In Lecture Notes in Computer in Economics and Mathematical Systems, Vol 206, springer Verlag 1983.

[Kohl67] D.A. Kohler. "Projection of Convex Polyhedral Sets". Ph.D. Thesis. University of California, Berkeley, 1967.

[LaLa91] C. Lassez and JL. Lassez. "Quantifier Elimination for Conjunctions of Linear Constraints via a Convex Hull Algorithm". To Appear 1991.

[LHMA89] JL. Lassez, T. Huynh and K. McAloon. "Simplification and Elimination of Redundant Arihtmetic Constraints". In *Proceedings of the North-American Conference on Logic programming (NACLP'89)*, Cleveland, Ohio, october 1989, MIT Press.

[Telg81] J. Telgen. "Redundancy and Linear Programming". Mathematical Center Tracts 137. Mathematisch Centrum, Amsterdam, 1981.

[Shri86] A. Schrijver. "Theory of Linear and Integer Programming". Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1987.

[VHen89] P. Van Hentenryck. "Constraint Satisfaction in Logic Programming". M.I.T. Press, 1989.

Domains decomposition in Finite Constraint-Satisfaction Problems*

Philippe Jégou
L.I.U.P. - Université de Provence
3, place Victor Hugo
F13331 Marseille cedex 3, France
jegou@gyptis.univ-mrs.fr

Abstract

In this paper, we present a method for improving search efficiency in the area of Constraint-Satisfaction-Problems in finite domains. This method is based on the analysis of the micro-structure of a CSP. We call micro-structure of a CSP, the graph defined by the compatible relations between variable-value pairs: vertices are these pairs, and edges are defined by pairs of compatible vertices. Given the micro-structure of a CSP, we can realize a pre-processing to simplify the problem with a decomposition of the domains of variables. So, we propose a new approach to problem decomposition in the field of CSPs, well adjusted in cases such as classical decomposition methods are without interest (i.e. when the constraint graph is complete). The method is described in the paper and a complexity analysis is presented, given theoretical justifications of the approach. Furthermore, a polynomial class of CSPs is induced by this approach, its recognition being linear in the size of the considered instance of CSP.

1 Introduction

Constraint-satisfaction problems (CSPs) involve the assignment of values to variables which are subject to a set of constraints. Examples of CSPs are map coloring, conjunctive queries in a relational databases, line drawings understanding, pattern matching in production rules systems, combinatorial puzzles... In the general case, finding a solution or testing if a CSP admits a solution is a NP-complete problem. A well known method for solving CSP is the Backtrack procedure. If n is the number of variables, d the size of the domains of variables, and m the number of constraints, the complexity of this procedure is $O(m.d^n)$. A better bound is given using decomposition methods as tree-clustering (Dechter & Pearl 1989) or cycle-cutset method (Dechter 1990). The complexity is then of the order of d^K, K being a parameter related to the structure of the CSP (the constraint graph). If the constraint network is a complete graph, then $K = n$. The decomposition methods are based on the structure of the CSP, i.e. the structure of the constraint graph.

In this paper, we present a decomposition method based on the "micro-structure" of the CSP. We call micro-structure of a CSP, the graph defined by the compatible relations between variable-value pairs: vertices are these pairs, and edges are defined by pairs of compatible vertices (compatible values). Given the graph associated to the micro-structure of a CSP, the problem of finding a solution to the CSP is equivalent to the problem of finding a n-clique (a set of vertices that induces a complete subgraph with these n vertices) in the micro-structure. Considering this property, we use triangulation of graphs (Kjaerulff 1990) and clustering of values driven by maximal cliques in the micro-structure to decompose the micro-structure associated to the CSP $\mathcal{P}$ to solve. This approach is motivated by the good algorithmic properties of triangulated graph, particularly to find maximal cliques. Every maximal clique induces a domains decomposition, and so, generates a collection of problems $\mathcal{P}_1, \mathcal{P}_2, \ldots \mathcal{P}_p$, equivalent to the initial problem $\mathcal{P}$. Each problem $\mathcal{P}_i$, corresponds to a sub-problem of $\mathcal{P}$ with a size of domains equal to δ_i, with the inequality $\delta_i \leq d$. So the complexity of solving $\mathcal{P}$ is now the sum of the complexities $O(m.\delta_i^n)$, for $i = 1, 2, \ldots p$. The complexity of the decomposition is linear in the size of the problem $\mathcal{P}$, and the number of new sub-problems is at most

*This work is supported by the BAHIA project of the PRC-GDR IA of CNRS.

linear in the size of $\mathcal{P}$. The quality of the decomposition is related to the value of each δ_i: more the value δ_i is small, more the decomposition is good. For example, if $\delta_i = 1$ or 2, the complexity of the problem $\mathcal{P}_i$ is now polynomial.

The section 2 introduces some preliminaries about CSPs while the third section defines formally the micro-structure. The method of domains decomposition is presented in the section 4. This is followed by a theoretical analysis of the method, concerning a complexity analysis, and showing polynomial classes of problems associated to the method.

2 Preliminaries

A finite CSP (Constraint Satisfaction Problem) is defined as a set X of n variables $X_1, X_2, \ldots X_n$, a set D of finite domains $D_1, D_2, \ldots D_n$, and a set C of m constraints $C_1, C_2, \ldots C_m$. A constraint C_i is defined on a set of variables $(X_{i_1}, \ldots, X_{i_j})$ by a subset of the cartesian product $(D_{i_1} \times D_{i_2} \ldots \times D_{i_j})$; we note this subset R_i (R_i specifies which values of the variables are compatible with each other). R is the set of all R_i, for $i = 1, 2, \ldots m$. So, we denote a CSP $\mathcal{P} = (X, D, C, R)$. A solution is an assignment of value to all variables which satisfies all the constraints. For a CSP $\mathcal{P}$, the hypergraph (X, C) is called the constraint hypergraph. A binary CSP is one in which all the constraints are binary, i.e. they involve only pairs of variables, so (X, C) is then a graph (called constraint graph) associated to (X, D, C, R). This paper deals only with binary CSPs. To simplify notations for binary CSPs, a constraint between variables X_i and X_j is denoted C_{ij}, and the associated relation R_{ij}. For a given CSP, the problem is either to find all solutions or one solution, or to know if there exists any solution; the last problem is known to be NP-complete.

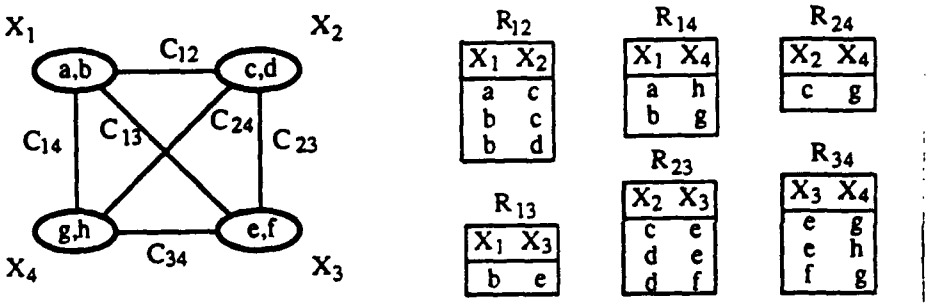


Figure 1. Binary CSP with complete constraint graph.

CSPs are normally solved by different versions of backtrack search. In this case, if d is the size of domains (maximum number of values in domains D_i), the theoretical time complexity of search is then $O(m.d^n)$. Consequently, many works try to improve the search efficiency. They mainly deal with binary CSPs. In (Freuder 1982), Freuder, considering the problem of finding one solution, gives a preprocessing procedure for selecting a good variable ordering prior to running the search. One of his main results is a sufficient condition for backtrack-free search. This condition concerns on one hand a structural property of the constraint graph, and on the other hand a local consistencies. After (Freuder 1982), Dechter and Pearl (Dechter and Pearl 1988) give two classes of polynomially solvable CSPs. For example, they define a property: if a binary CSP is arc-consistent, and if its constraint graph is acyclic, then the CSP admits a solution and there is a backtrack-free search order. This property holds for n-ary CSPs with hypergraphs (Janssen et al 1989).

Some methods use decomposition techniques based on structural properties of the CSP. These methods exploit the fact that the tractability of CSPs is intimately connected to the topological structure of their underlying constraint graphs. Moreover, these methods give an upper bound to the complexity of the problem, therefore, an upper bound to the search. The above property gives the goal of the transformation: given a CSP, the result must be an other CSP, equivalent to the first one, whose the structure is a tree. Two methods are based on this principle: the cycle-cutset method (Decther 1990) and tree-clustering scheme (Decther & Pearl 1989).

The cycle-cutset method (CCM) is based on the notion of cycle-cutset. The cycle-cutset of a graph, is a set of vertices such as the deletion of these vertices induces an acyclic graph. CCM is based on the fact that variables assignments changes the effective connectivity of the constraint graph. So, as soon as all the variables of the cycle-cutset are assigned, all the cycles of the constraint graph are cut. Therefore, the resulting problem is tree-structured and Freuder's theorem (Freuder 1982) can be applied to solve it. A property summarizes the method: if all the variables belonging to the cycle-cutset are instanciated, and if the resulting CSP is arc-consistent, then the problem admits solutions and a backtrack-free order. So, searching a solution, we can consider that the size of cycle-cutset corresponds to the height of the backtracking. More precisely, if K is the size of the cycle-cutset, the complexity of CCM is $O(m.d^{K+2})$.

Tree-clustering (TC) consists in forming clusters of variables such as the interactions between the clusters are tree structured. The hyper-edges of the induced constraint hypergraph are defined by the clusters of variables. The new CSP is equivalent to the first one, but the associated constraint hypergraph is acyclic. So, the property concerning acyclic n-ary CSPs holds for this CSP. If E is the size of the maximal cluster, the complexity of TC is then $O(n.E.d^E)$.

If the constraint network is a complete graph, we have the equality $E = K+2 = n$. So, the complexity of decomposition methods is the same than for classical backtracking, of the order of d^n. Consequently, complete constraint graphs (n-cliques) can be considered as hard instances of CSP for decomposition methods. The decomposition method described in this paper proposes a solution to handle these hard CSPs, but can also be used on incomplete constraints graph. It is based on a decomposition of the micro-structure of a CSP.

3 Microstructure of CSPs

We call micro-structure of a CSP, the graph defined by the compatible relations between variable-value pairs: vertices are these pairs, and edges are defined by pairs of compatible vertices.

Definition 1. Given a binary CSP $\mathcal{P} = (X, D, C, R)$ such as (X, C) is a complete graph, $\mu(\mathcal{P})$ is called *micro-structure* of $\mathcal{P}$ and it is a n-partite graph defined by:

- $X_D = \{(X_i, a)/X_i \in X \text{ and } a \in D_i\}$
- $C_R = \{\{(X_i, a), (X_j, b)\}/(X_i, X_j) = C_{ij} \in C \text{ and } (a, b) \in R_{ij}\}$
- $\mu(\mathcal{P}) = (X_D, C_R)$

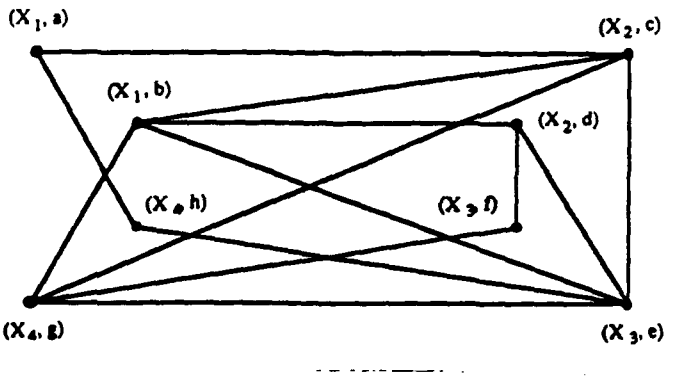


Figure 2. Micro-structure of the CSP given in figure 1.

Necessary, $\mu(\mathcal{P})$ is a n-partite graph because it can not exist edges between vertices of a same domain. In the example in figure 2, we have sets $\{(X_1, a), (X_1, b)\}, \{(X_2, c), (X_2, d)\}, \{(X_3, e), (X_3, f)\}$ and $\{(X_4, g), (X_4, h)\}$ with no one edge between vertices associated to the same variable.

If (X, C) is not a complete graph, i.e. there are two variables X_i and X_j such as the constraint C_{ij} does not exist between variables X_i and X_j, $\mu(\mathcal{P})$ can be completed adding the universal relation between these variables. The universal relation is the relation $R_{ij} = D_i \times D_j$ (all pairs of values are compatible). In this paper we always consider CSPs with complete constraint graph.

Given a CSP $\mathcal{P} = (X, D, C, R)$ and its micro-structure $\mu(\mathcal{P})$, we can derive a basic property.

Property 2. Given a CSP $\mathcal{P} = (X, D, C, R)$ and its micro-structure $\mu(\mathcal{P})$ we have:

$$(a_1, a_2, \ldots a_n) \text{ is a solution of } \mathcal{P} \Leftrightarrow \{(X_1, a_1), (X_2, a_2), \ldots (X_n, a_n)\} \text{ is a n-clique of } \mu(\mathcal{P})$$

Proof. $(a_1, a_2, \ldots a_n)$ *is a solution of* $\mathcal{P} \Leftrightarrow \forall i, j, 1 \leq i < j \leq n, (a_i, a_j) \in R_{ij} \Leftrightarrow \forall i, j, 1 \leq i < j \leq n, \{(X_i, a_i), (X_j, a_j)\} \in C_R \Leftrightarrow \{(X_1, a_1), \ldots (X_n, a_n)\}$ *is a n-clique of* $\mu(\mathcal{P})$

We remark that a solution of $\mathcal{P}$ corresponds to a covering of n vertices in the constraint graph (X, C): there is exactly one vertex (X_i, a) for each domain D_i, for $i = 1, 2, \ldots n$. So, solving a CSP can be considered as the problem of finding a n-clique in its micro-structure. The method we present for the decomposition of domains is based on the topological analysis of the microstructure, related to the existence of n-cliques.

4 Simplifying CSPs by domains decomposition

We seen that solving a CSP (finding one solution) can be considered as the problem of finding a *n*-clique in its microstructure. This problem is known to be NP-hard (Karp 1972), but there are classes of graphs such as polynomial (linear) algorithms have been defined. The method we present is based on one of these classes: triangulated graphs. So, some definitions and properties must be recalled.

Definition 3. A graph is *triangulated* iff every cycle of length at least four has a chord, i.e. an edge joining two non-consecutive vertices along the cycle.

Property 4. (Fulkerson & Gross 1965) A triangulated graph on n vertices has at most n maximal cliques (a clique is maximal iff it is not included in an other clique).

Property 5. (Gavril 1972) The problem of finding all maximal cliques in a triangulated graph (X, C) is in $O(N + M)$ if $N = \mid X \mid$ and $M = \mid C \mid$.

Given the micro-structure of any CSP, it is not possible to immediately use these properties because any micro-structure is not necessary a triangulated graph (eg. the micro-structure in the figure 2).

Nevertheless, it is possible to use these results: given any graph $G = (X, C)$, it is possible to add new edges in C to obtain C', such as the graph $\Gamma(G) = (X, C')$ is a triangulated graph. This addition of edges is called *triangulation*, and can be realized in a linear time in the size of the graph (Kjaerulff 1990).

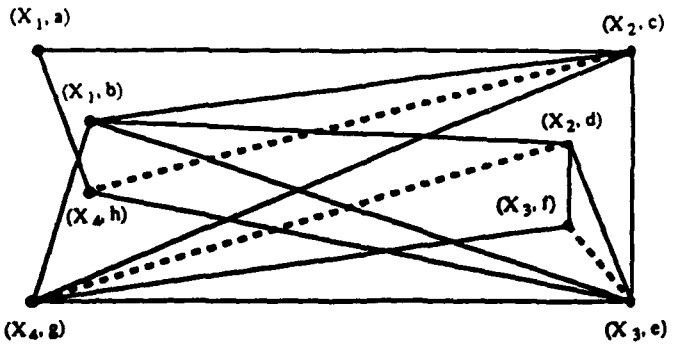


Figure 3. Triangulation of the micro-structure of figure 2. Added edges are given by the dotted lines.

After a triangulation, it is possible to apply the property 5. Suppose we have a CSP $\mathcal{P} = (X, D, C, R)$ and its micro-structure $\mu(\mathcal{P}) = (X_D, C_R)$. Consider a triangulated graph defined by a triangulation of (X_D, C_R). There are three classes of edges in $\Gamma(X_D, C_R)$:

- edges $\{(X_i, a), (X_j, b)\}$ already in $\mu(\mathcal{P})$. i.e. (a, b) is in R_{ij}

- edges $\{(X_i, a), (X_j, b)\}/i \neq j$: adding this edge corresponds to add the tuple (a, b) in R_{ij}
- edges $\{(X_i, a), (X_i, b)\}$: adding this edge has no semantic

Since $\Gamma(X_D, C_R)$. is a triangulated graph. we know that there are no more than $n.d$ maximal cliques in this graph (by property 4), and that it is possible to find them with a linear algorithm (by property 5). Furthermore, we know that if there exists solutions, anyone is in a maximal clique of this triangulated graph, and consequently, the search of solutions of $\mathcal{P}$ will be limited to the search of solutions on separated problems, each one associated to a maximal clique.

Consider Y, a maximal clique in the triangulated graph $\Gamma(X_D, C_R)$; two possibilities must be considered:

- Y is not a covering of all domains: there is at least one variable X_i of X that does not appear in the vertices (X_i, a) of Y. Consequently. the clique Y does not contain a n-clique that is a covering of all domains, and so there is no solution in Y.
- Y is a covering of all domains. Given Y, we can induce a new CSP, by the projection of vertices in Y on each domains. So, we obtain a collection of domains $D_{Y,i}$ such as $D_{Y,i} \subseteq D_i$, each new domain $D_{Y,i}$ being induced by the vertices (X_i, a) in Y. The constraints of the new CSP associated to Y are the old constraints. restricted to the values in new domains. Searching a solution can be realized on this new CSP. Nevertheless. the fact that Y is a covering of all domains does not guarantee that there is a solution. because the triangulation adds some new edges that connect vertices corresponding to incompatible values.

Theoretical foundations of the method are given below.

Definition 6. Given a binary CSP $\mathcal{P} = (X, D, C, R)$. its micro-structure $\mu(\mathcal{P}) = (X_D, C_R)$. and Y a subset of X_D. The CSP induced by Y on $\mathcal{P}$. denoted $\mathcal{P}(Y)$ is defined by:

- $D_Y = \{D_{Y,1}, \ldots D_{Y,n}\}$ such as $D_{Y,i} = \{a \in D_i / (X_i, a) \in Y\}$
- $R_{Y,ij} = \{(a, b) \in R_{ij} / (X_i, a), (X_j, b) \in Y\}$
- $\mathcal{P}(Y) = (X, D_Y, C, R_Y)$

The theorem below define the principle driving the domains decomposition:

Theorem 7. Given a binary CSP $\mathcal{P} = (X, D, C, R)$. its micro-structure $\mu(\mathcal{P})$ and $Y = \{Y_1, \ldots Y_p\}$, the set of the maximal cliques of $\Gamma(\mu(\mathcal{P}))$. we have:

$$Solutions(\mathcal{P}) = \cup_{i=1}^{p} Solutions(\mathcal{P}(Y_i))$$

Proof.

- *With property 2, we know that any solution of the problem $\mathcal{P}$ is associated to a n-clique. So, this n-clique is necessary included in one set Y_i because in a graph. each clique belongs necessary to a maximal clique of the considered graph. Consequently. the considered solution of $\mathcal{P}$ is necessary a solution of $\mathcal{P}(Y_i)$.*
- *Every solution of a problem $\mathcal{P}(Y_i)$ is a clique in $\mu(\mathcal{P})$ because all the edges of this clique are edges induced by compatible values in $\mathcal{P}$. Consequently, every solution of $\mathcal{P}(Y_i)$ is a solution of $\mathcal{P}$.*

We remark that a solution of $\mathcal{P}(Y_i)$ can appear as a solution of an other $\mathcal{P}(Y_i)$. In the figure 4, we present the applying of theorem 7 to the example.

Maximal cliques	Decomposed domains
$Y_1 = \{(X_1, a), (X_2, c), (X_4, h)\}$	$D_{Y_1,1} = \{a\}, D_{Y_1,2} = \{c\}, D_{Y_1,3} = \emptyset, D_{Y_1,4} = \{h\}$
$Y_2 = \{(X_1, b), (X_2, c), (X_3, e), (X_4, g)\}$	$D_{Y_2,1} = \{b\}, D_{Y_2,2} = \{c\}, D_{Y_2,3} = \{e\}, D_{Y_2,4} = \{g\}$
$Y_3 = \{(X_1, b), (X_2, d), (X_3, e), (X_3, f)\}$	$D_{Y_3,1} = \{b\}, D_{Y_3,2} = \{d\}, D_{Y_3,3} = \{e, f\}, D_{Y_3,4} = \emptyset$
$Y_4 = \{(X_2, c), (X_3, e), (X_4, h)\}$	$D_{Y_4,1} = \emptyset, D_{Y_4,2} = \{c\}, D_{Y_4,3} = \{e\}, D_{Y_4,4} = \{h\}$
$Y_5 = \{(X_1, b), (X_3, e), (X_3, f), (X_4, g)\}$	$D_{Y_5,1} = \{b\}, D_{Y_5,2} = \emptyset, D_{Y_5,3} = \{e, f\}, D_{Y_5,4} = \{g\}$

Figure 4. Applying theorem 7 to the CSP of figure 1. The cliques Y_1, Y_3, Y_4 and Y_5 do not cover all the domains; so the induced sub-problem are not consistent. On the other hand, the cliques Y_2 induces a consistent sub-problem.

Algorithm:

- **generation of $\mu(\mathcal{P})$**
- **triangulation of $\mu(\mathcal{P})$; we obtain $\Gamma(\mu(\mathcal{P}))$**
- **research of all maximal cliques in $\Gamma(\mu(\mathcal{P}))$; the result of this step is $Y = \{Y_1, \ldots Y_p\}$**
- **for all Y_i in Y do**
  if Y_i is a covering of all the domains in D
  then solve $\mathcal{P}(Y_i)$ else $\mathcal{P}(Y_i)$ has no solution

The first step is realized first with an enumeration of the values of all the domains to obtain the vertices of $\mu(\mathcal{P})$, and secondly, with an enumeration of all the compatible tuples of relations to obtain the edges of $\mu(\mathcal{P})$. If the problem $\mathcal{P}$ has not a complete constraint graph, it is possible to transform it with the addition of the universal constraint between non-connected variables. The second step can be realized using triangulation algorithms - see (Kjaerulff 1990). The maximal cliques can be obtained by the algorithm of Gavril (Gavril 1972)(Golumbic 1980). The last step is first realized with the generation of the problem $\mathcal{P}(Y_i)$: it is sufficient to define new domains based on the vertices in Y_i. Finally, solving $\mathcal{P}(Y_i)$ is possible with any classical method such as standard backtracking for example.

5 Theoretical analysis

We first give some notations. Given $\mathcal{P} = (X, D, C, R)$ and its micro-structure $\mu(\mathcal{P}) = (X_D, C_R)$.

- n is the number of variables
- d is the maximal number of values in domains, i.e. $\forall i, 1 \leq i \leq n, \mid D_i \mid \leq d$
- $\mathcal{N}$ the number of vertices in $\mu(\mathcal{P})$: $\mathcal{N} = \sum_{i=1}^{n} \mid D_i \mid \leq n.d$
- m is the number of constraints; if the constraint grapgh is complete, then $m = n.(n-1)/2$.
- M is the number of edges in $\mu(\mathcal{P})$: $M = \sum_{i,j=1}^{n} \mid R_{ij} \mid \leq \mathcal{N}.(\mathcal{N}-1)/2 < n^2.d^2$
- p is the number of maximal cliques in $\Gamma(\mu(\mathcal{P}))$; $p \leq n.d$.

The cost of step 1 in the algorithm is $O(\mathcal{N} + M)$. Nevertheless, if (X, C) is not a complete graph, we have $O(n^2.d^2)$. Triangulation step (step 2) is linear is the size of the resultant graph: $O(\mathcal{N} + M')$, if M' is the new set of edges after triangulation. Necessary, $M \leq M' < n^2.d^2$. The cost of finding all maximal cliques in $\Gamma(\mu(\mathcal{P}))$ is also linear: $O(\mathcal{N} + M')$. By property 4, we know that the number of maximal cliques p satisfies the inequality $p \leq \mathcal{N}$.

For the last step, we first evaluate the cost of solving one problem $\mathcal{P}(Y_j)$; it can be bounded by:

$$O(m.(\prod_{i=1}^{n} \mid D_{Y_j,i} \mid))$$

So, the cost of the last step, i.e. the cost of solving all sub-problems $\mathcal{P}(Y_j)$, for $j = 1, 2, \ldots p$, is:

$$O(m.(\sum_{j=1}^{p}(\prod_{i=1}^{n} \mid D_{Y_{j,i}} \mid)))$$

The comparison of this cost with respect to the cost of standard backtracking on the initial problem is necessary. The cost of backtracking on $\mathcal{P}$ is

$$O(m.(\prod_{i=1}^{n} \mid D_i \mid))$$

If we consider $d = \mid D_i \mid$ and $\delta = \mid D_{Y_{j,i}} \mid$, for $i = 1, 2, \ldots n$ and $j = 1, 2, \ldots p$, the comparison between standard backtracking and domains decomposition is now

$$m.d^n \textbf{ VS } m.p.\delta^n$$

or

$$d^n \textbf{ VS } p.\delta^n$$

We know that p is bounded by $n.d$ (cf. property 4). So we give comparison of exponential terms, d^n and δ^n. Suppose that the decomposition induces a simplification of domains, such as we have for example $d = 2.\delta$. The comparison is now

$$d^n \textbf{ VS } \frac{n.d}{2^n}.\delta^n$$

because $p.\delta^n = p.(d/2)^n = p.(1/2)^n.d^n \leq [n.d.(1/2)^n].d^n$.

Consequently, the decomposition can be very interesting on the instances of problems such as these kind of hypothesis on d and δ hold, i.e. for the problems such as we have $[n.d.(1/2)^n] \ll 1$.

The decomposition method induces a polynomial class of CSP. If we consider the class of CSPs $\mathcal{P}$ such as the triangulation of their micro-structure $\mu(\mathcal{P})$ connects at most two values belonging to the same domain in every obtained maximal cliques, this class of CSPs is polynomial:

Property 8. Let $\mathcal{P}$ be a CSP, and its micro-structure $\mu(\mathcal{P})$. If in $\Gamma(\mu(\mathcal{P}))$ there is at most one new edge $\{(X_i, a), (X_i, b)\}$ per domain D_i in every new maximal cliques then, there is a polynomial algorithm to solve $\mathcal{P}$ (searching for one solution).

Proof. *After applying the algorithm for triangulation of the micro-structure $\mu(\mathcal{P})$, the size of domains in all the induced sub-problems $\mathcal{P}(Y_j)$ is at most two. Consequently, all induced sub-problems can be solved applying the result given in (Dechter 1992). One corollary of this theorem deals for binary CSPs with bivalent domains, and provides a polynomial method to solve this class of CSPs.*

The interest of this class is that checking for the adherence will be linear in the size of any checked instance. Nevertheless, a real problem concerning this class is to the kind of constraints that we can represent satisfying this kind of property.

6 Conclusion

We proposed a new method to reduce domains in constraint satisfaction problems. This method is based on the analysis of the micro-structure of CSP, i.e. the structure of the relations between compatibles values of the domains. Given the micro-structure of a CSP, we present a scheme to decompose domains of variables, forming a set of sub-problems such as they have necessary less values than domains in the initial problem. This decomposition is driven by combinatorial and algorithmic properties of triangulated graphs. The complexity analysis of the method shown the theoretical advantages of the approach. Indeed, given a CSP $\mathcal{P}$, if d is the size of domains of the n variables, and if this problem is defined on m constraints, the complexity of any search like standard backtracking, is $O(m.d^n)$. We shown that the method induces the complexity $O(m.p.\delta^n)$ with p being the number of induced sub-problems - p is necessary linear in the size of the problem

$\mathcal{P}$ - and δ is the size of new domains, always satisfying $\delta \leq d$. Furthermore, a polynomial class of CSPs has been defined, the recognition of its elements being linear in the size of instances.

The decomposition method is at present only defined on binary CSPs. Nevertheless, an extension to n-ary CSPs is possible. A way to realize this extension consists in using primal constraint graph (Dechter & Pearl 88). Suppose we have a n-ary CSP with a constraint C_l between three variables; that is $C_l = \{X_i, X_j, X_k\}$. To generate the microstructure, we consider three binary constraints: C_{ij}, C_{ik} and C_{jk}. The associated relations are $R_{ij} = R_l[(X_i, X_j)], R_{ik} = R_l[(X_i, X_k)]$ and $R_{jk} = R_l[(X_j, X_k)]$. This primal representation is not equivalent to the initial n-ary CSP because the new problem is less constrained. But it is sufficient to realize domains decomposition, since the constraints finally considered to solve the initial CSP will be the initial n-ary constraints, with possibly, smallest domains.

Now, an experimental analysis must be realized to see practical interests of the approach.

References

Dechter, R. 1990. Enhancement Schemes for Constraint-satisfaction problems: Backjumping, Learning and Cutset Decomposition. Artificial Intelligence 41:273-312.

Dechter, R. & Pearl, J. 1988. Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence 34:1-38.

Dechter, R. & Pearl, J. 1989. Tree Clustering for Constraint Networks. Artificial Intelligence 38:353-366.

Freuder, E.C. 1982. A sufficient condition for backtrack-free search. JACM, 29(1):24-32.

Fulkerson, D.R. & Gross, O. 1965. Incidence matrices and interval graphs. Pacific J. Math. 15:835-855.

Gavril, F. 1972. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. SIAM J. Comput 1(2):180-187.

Golumbic, M.C. 1980. Algorithmic Graph Theory and Perfect Graphs. Academic Press. New-York.

Janssen, P., Jégou, P., Nouguier, B. & Vilarem, M.C. 1989. A filtering process for general constraint satisfaction problems: achieving pairwise-consistency using an associated binary representation. In Proceedings of the IEEE Workshop on Tools for Artificial Intelligence, 420-427. Fairfax, USA.

Karp, E.C. 1972. Reducibility among combinatorial problems. In Complexity of Computer Computation. 85-103. Miller & Thatcher Eds. Plenum Press. New-York.

Kjaerulff, U. 1990. Triangulation of Graphs - Algorithms Giving Small Total State Space. Judex R.R. Aalborg. Denmark.

Memoization in Constraint Logic Programming*

Mark Johnson
Department of Cognitive and Linguistic Sciences, Box 1978
Brown University
mj@cs.brown.edu

Abstract

Motivated by a natural language processing application, this paper shows how to extend memoization techniques for logic programs to constraint logic programming. The *lemma table proof procedure* presented here generalizes standard memoization proof procedures such as OLDT resolution by (i) allowing goals and constraints to be resolved in any order, (ii) permitting memoization on sets of goals and constraints rather than only individual goals, and (iii) allowing the solutions recorded in the memo table to include unresolved goals and constraints, which are "inherited" by the calling routine.

1 Introduction

This paper shows how to apply memoization (caching of subgoals and associated answer substitutions) in a constraint logic programming setting. The research is is motivated by the desire to apply constraint logic programming (CLP) to problems in natural language processing.

In general, logic programming provides an excellent theoretical framework for computational linguistics [13]. CLP extends "standard" logic programming by allowing program clauses to include constraints from a specialized contraint language. For example, the CLP framework allows the feature-structure constraints that have proven useful in computational linguistics [15] to be incorporated into logic programming in a natural way [1, 5, 16].

Because modern linguistic theories describe natural language syntax as a system of interacting "modules" which jointly determine the linguistic structures associated with an utterance [2], a grammar can be regarded as a conjunction of constraints whose solutions are exactly the well-formed or grammatical analyses. Parsers for such grammars typically coroutine between a tree-building component that generates nodes of the parse tree and the well-formedness constraints imposed by the linguistic modules on these tree structures [4, 6, 8]. Both philosophically and practically, this fits in well with the CLP approach.

But the standard CLP framework inherits some of the weaknesses of the SLD resolution procedure that it is based on. When used with the standard formalization of a context-free grammar the SLD resolution procedure behaves as a recursive descent parser. With left-recursive grammars such parsers typically fail to terminate because a goal corresponding to a prediction of a left-recursive category can reduce to an identical subgoal (up to renaming), producing an "infinite loop". Standard techniques for left-recursion elimination [12, 13] in context-free grammars are not always directly applicable to grammars formulated as the conjunction of several constraints [10].

With memoization, or the caching of intermediate goals (and their corresponding answer substitutions), a goal is solved only once and its solutions are cached; the solutions to identical goals are obtained from this cache. Left recursion need not lead to non-termination because identical subgoals are not evaluated, and the infinite loop is avoided. Further, memoization can sometimes provide the advantages of dynamic programming approaches to parsing: the Earley deduction proof procedure (a memoized version of SLD resolution) simulates an Earley parse [3] when used with the standard formalization of a context-free grammar [14].

*This research was initiated during a summer visit to the IMSV, Universität Stuttgart; which I would like to thank for their support. Thanks also to Pascal van Hentenryck and Fernando Pereira for their important helpful suggestions.

```
parse(String, Tree) :- wf(Tree, s), y(Tree, String, []).

y(_-Word, [Word|Words], Words).
y(_/[Tree1], Words0, Words) :- y(Tree1, Words0, Words).
y(_/[Tree1,Tree2], Words0, Words) :-
    y(Tree1, Words0, Words1), y(Tree2, Words1, Words).

wf(np-kim, np).                                   % NP → Kim
wf(n-friend, n).                                  % N → friend
wf(v-walks, v).                                   % V → walks
wf(s/[Tree1, Tree2], s) :- wf(Tree1, np), wf(Tree2, vp).   % S → NP VP
wf(np/[Tree1, Tree2], np) :- wf(Tree1, np), wf(Tree2, n).  % NP → NP N
wf(vp/[Tree1], vp) :- wf(Tree1, v).               % VP → V
```

Figure 1: A grammar fragment

Thus constraint logic programming and memoization are two recent developments in logic programming that are important for natural language processing. But it is not obvious how, or even if, the two can be combined in a single proof procedure. For example, both Earley Deduction [14] and OLDT resolution [17, 20] resolve literals in a strict left-to-right order, so they are not capable of rudimentary constraint satisfaction techniques such as goal delaying. The strict left-to-right order restriction is relaxed but not removed in [18, 19], where literals can be resolved in any local order. This paper describes soundness and completeness proofs for a proof procedure that extends these methods to allow for goal delaying. In fact, the lemma table proof procedure generalizes naturally to constraint logic programming over arbitrary domains, as described below.

The lemma table proof procedure generalizes Earley Deduction and OLDT resolution in three ways.

- Goals can be resolved in any order (including non-local orders), rather than a fixed left-to-right or a local order.
- The goals entered into the table consist of non-empty sets of literals rather than just single literals. These sets can be viewed as a single program literal and zero or more constraints that are being passed down into the subsidary proof.
- The solutions recorded in the lemma table may contain unresolved goals. These unresolved goals can be thought of as constraints that are being passed out of the subsidary proof.

2 A linguistic example

Consider the grammar fragment in Figure 1 (cf. also [4, pages 142–177]). The parse relation holds between a string and a tree if the yield of the tree is the string to be parsed and tree satisfies a well-formedness condition. In this example, the well-formedness condition is that the tree is generated by a simple context-free grammar, but in more realistic fragments the constraints are considerably more complicated.

Trees are represented by terms. A tree consisting of a single pre-terminal node labelled C whose single child is the word W is represented by the term C-W. A tree consisting of a root node labelled C dominating the sequence of trees $T_1 \ldots T_n$ is represented by the term $C/[T_1, \ldots, T_n]$.

The predicate wf(*Tree, Cat*) holds if *Tree* represents a well-formed parse tree with a root node labelled *Cat* for the context free grammar shown in the comments. The predicate y(*Tree, S0, S*) holds if *S0–S* is a "difference list" representing the yield of *Tree*; it collects the terminal items in the familiar tree-walking fashion. From this program, the following instances of parse can be deduced (these are meant to approximate possessive constructions like *Kim's friend's friend walks*).

```
parse([kim,walks], s/[np-kim,vp/[v-walks]]).
parse([kim,friend,walks], s/[np/[np-kim,n-friend],vp/[v-walks]]).
```

(1) parse(KW,T).
(2) wf(T,s), y(T,KW,[]).
(3) wf(_/[T1,T2],s), y(T1,KW,S1), y(T2,S1,[]).
(4) wf(T1,np), wf(T2,vp), y(T1,KW,S1), ...
(5) y(T3,KW,S3), y(T4,S3,S1), wf(_/[T3,T4],np), ...
(6) wf(T3,np), wf(T4,n), y(T3,KW,S3), ...
(7) y(T5,KW,S5), y(T6,S5,S3), wf(_/[T5,T6],np), ...
...

Figure 2: An infinite SLD refutation, despite co-routining

parse([kim,friend,friend,walks], s/[np/[np/[np-kim,n-friend],n-friend],vp/[v-walks]]).
...

The parsing problem is encoded as a goal as follows. The goal consists of an atom with the predicate parse whose first argument instantiated to the string to be parsed and whose second argument is uninstantiated. The answer substitution binds the second argument to the parse tree. For example, an answer substitution for the goal parse([Kim,friend,walks], Tree) will have the parse tree for the string *Kim friend walks* as the binding for Tree.

Now consider the problem of parsing using the program shown in Figure 1. Even with the variable String instantiated, both of the subgoals of *parse* taken independently have an infinite number of subsumption-incomparable answer substitutions. Informally, this is because there are an infinite number of trees generated by the context-free grammar, and there are an infinite number of trees that have any given non-empty string as their yield. Because the standard memoization techniques mentioned above all compute all of the answer substitutions to every subgoal independently, they never terminate on such a program.

However, the set of answer substitions that satisfy both constraints is finite, because the number of parse trees with the same yield with respect to this grammar is finite. The standard approach to parsing with such grammars takes advantage of this by using a selection rule that "coroutines" the goals wf and y, delaying all wf goals until the first argument is instantiated to a non-variable. In such a system, the wf goals function as constraints that filter the trees generated by the goals y.

In this example, however, there is a second, related, problem. Coroutining is not sufficient to yield a finite SLD tree, even though the number of refutations is finite. Informally, this is because the grammar in Figure 1 is left recursive, and the search space for a recursive descent parser (which an SLD refutation mimics with such a program) is infinite.

Figure 2 shows part of an infinite SLD derivation from the goal parse(KW, T), where KW is assumed bound to [kim,walks] (although the binding is actually immaterial, as no step in this refutation instantiates this variable). The selection rule expresses a "preference" for goals with certain arguments instantiated. If there is a literal of the form wf(T, C) with T instantiated to a non-variable then the left-most such literal is selected, otherwise if there is a literal of the form y($T, S0, S$) with $S0$ instantiated to a non-variable then the left-most such literal is selected, otherwise the left-most literal is selected. The selected literal is underlined, and the new literals introduced by each reduction are inserted to the left of the old literals.

Note that the y and wf literals resolved at steps (6) and (7) in Figure 2 are both children of and variants of the literals resolved at steps (5) and (6). This sequence of resolution steps can be iterated an arbitrary number of times. It is a manifestation of the left recursion in the well-formedness constraint wf.

One standard technique for dealing with such left-recursion is memoization [14, 13]. But there are two related problems in applying the standard logic programming memoization techniques to this problem.

First, because the standard methods memoize and evaluate at the level of an individual literal, the granularity at which they apply memoization is is to small. As noted above, in general individual wf or y literal can have an infinite number of answer substitutions. The lemma table proof procedure circumvents this problem by memoizing *conjunctions* of literals (in this example, a conjunction of wf and *y* literals which has only a finite number of subsumption-incomparable valid instances).

The second problem is that the standard memoization techniques restrict the order in which literals can be resolved. In general, these restrictions prevent the "goal delaying" required to co-routine among several

constraints. The lemma table proof procedure lifts this restriction by allowing arbitrary selection rules.

3 The Lemma Table proof procedure

Like the Earley Deduction and the OLDT proof procedures, the Lemma Table proof procedure maintains a lemma table that records goals and their corresponding solutions. After a goal has been entered into the lemma table, other occurences of instances of that goal can be reduced by the solutions from the lemma table instead of the original program clauses.

We now turn to a formal presentation of the Lemma Table proof procedure. In what follows, lower-case letters are used for variables that range over atoms. Upper-case letters are used for variables that range over *goals*, which are sets of atoms. Goals are interpreted conjunctively; a goal is satisfied iff all of its members are.

A goal G *subsumes* a goal G' iff there is some substitution θ such that $G' = G\theta$. (Note that m.g.u.'s for sets of goals are in general not unique even up to renaming).

The "informational units" manipulated by the lemma table proof procedure are called generalized clauses. A *generalized clause* is a pair of goals, and is written $G_1 \leftarrow G_2$. G_1 is called the *head* of the clause and G_2 is called the *body*. Both the head and body are interpreted conjunctively; i.e., $G_1 \leftarrow G_2$ should be read as "if each of the G_2 are true, then all of the G_1 are true". A generalized clause has a natural interpretation as a goal subject to constraints: G_1 is true in any interpretation which satisfies the constraints expressed by G_2.

A *lemma table* is a set of *table entries* of the form $\langle G, T, S \rangle$, where

1. G is a goal (this entry is called a table entry *for* G),
2. T is a lemma tree (see below), and
3. S is a sequence of clauses, called the *solution list* for this entry.

A *lemma tree* is a tree constructed by the algorithm described below. Its nodes have two labels. These are

1. a clause $A \leftarrow B$, called the *clause labelling* of the node, and
2. an optional tag, which when present is one of solution, program(b) for some $b \in B$, or table(B', p) where $\emptyset \subset B' \subseteq B$ and p is either the null pointer nil or a pointer into a solution list of a table entry for some G that subsumes B'.

Untagged nodes are nodes that have not yet been processed. All nodes are untagged when they are created, and they are assigned a tag as they are processed. The tags indicate which kind of resolution has been applied to this clause. A node tagged program(b) is resolved against the clauses defining b in the program. A node tagged table(B', p) is resolved against the instances of a table entry E for some goal that subsumes B'; the pointer p keeps track of how many of the solutions from E have been inserted under this node (just as in OLDT resolution). Finally, a node tagged solution is not resolved, rather its clause labelling is added to the solution list for this table entry.

Just as SLD resolution is controlled by a selection rule that determines which literal will be reduced next, the Lemma Table proof procedure is controlled by a control rule R which determines the next goal (if any) to be reduced and the manner of its reduction.

More precisely, R must tag a node with clause labelling $A \leftarrow B$ with a tag that is either solution, program(b) for some $b \in B$, or table(B', nil) such that $\emptyset \subset B' \subseteq B$. Further, R must tag the root node of every lemma tree with the tag program(b) for some b (this ensures that some program reductions are performed in every lemma tree, and hence that a lemma table entry cannot be used to reduce itself vacuously).

Finally, as in OLDT resolution, the Lemma Table proof procedure allows a user-specified *abstraction operation* α that maps goals to goals such that $\alpha(G)$ subsumes G for all goals G. This is used to generalize the goals in the same way as the term-depth abstraction operation in OLDT resolution, which it generalizes.

The Lemma Table proof procedure can now be presented.

Input: A non-empty goal G, a program P, an abstraction operation α, and a control rule R.

Output: A set γ of clauses of the form $G' \leftarrow C$, where G' is an instance of G.

Algorithm: Create a lemma table with one table entry $\langle G, T, [] \rangle$, where T contains a single untagged node with the clause labelling $G \leftarrow G$. Then repeat the following operations until no operation applies. Finally, return the solution list from the table entry for G.

The operations are as follows.

Prediction Let v be an untagged node in a lemma tree T of table entry $\langle G, T, S \rangle$, and let v's clause labelling be $A \leftarrow B$. Apply the rule R to v, and perform the action specified below depending on the form of the tag R assigned to v.

solution : Add $A \leftarrow B$ to the end of the solution list S.

program(b) : Let $B' = B - \{b\}$. Then for each clause $b' \leftarrow C$ in P such that b and b' unify with a m.g.u. θ, create an untagged child node v' of v labelled $(A \leftarrow B' \cup C)\theta$.

table(B', nil) : The action in this case depends on whether there already is a table entry $\langle G', T', S' \rangle$ for some G' that subsumes B'. If there is, set the pointer in the tag to the start of the sequence S'. If there is not, create a new table entry $\langle \alpha(B'), T'', [] \rangle$, where T'' contains a single untagged node with clause labelling $\alpha(B') \leftarrow \alpha(B')$. Set the pointer in v's tag to point to the empty solution list of this new table entry.

Completion Let v be a node with clause labelling $A \leftarrow B$ and tagged table(B', p) such that p points to a non-null portion S' of a solution list of some table entry. Then advance p over the first element $B'' \leftarrow C$ of S' to point to the remainder of S'. Further, if B' and B'' unify with m.g.u. θ, then add a new untagged child node to v labelled $(A \leftarrow (B - B') \cup C)\theta$.

It may help to consider an example based on the program in Figure 1. The computation rule R used is the following. Let v be a node in a lemma tree and let $A \leftarrow B$ be its clause label. If v is the root of a lemma tree and B contains a literal of the form y($T, S0, S$) then $R(v)$ = program(y($T, S0, S$)). If B is empty then $R(v)$ = solution (no other tagging is possible for such nodes). If B contains a literal of the form wf(T, C) where T is a non-variable then $R(v)$ = program(wf(T, C)) (there is never more than one such literal). Otherwise, if B contains two literals of the form wf(T, C), y($T, S0, S$) where $S0$ is a non-variable, then $R(v)$ = table({wf(T, C), y($T, S0, S$)}, nil). These four cases exhaust all of the node labelling encountered in the example.[1]

Figure 3 depicts the completed lemma table constructed using the rule R for the goal wf(Tree,s), y(Tree,[kim,walks],[]). To save space, kim and walks are abbreviated to k and w respectively. Only the tree from each table entry is shown because the other components of the entry can be read off the tree. The goal of each table entry is the head of the clause labelling its root clause, and is shown in bold face. Solution nodes are shown in italic face. Lookup nodes appear with a dashed line pointing to the table entry used to reduce them.

The first few steps of the proof procedure are the following; each step corresponds to the circled node of the same number.

(1) The root node of the first table entry's tree restates the goal to be proven. Informally, this node searches for an S located at the beginning of the utterance. The literal y(Tree,[kim,walks],[]) is selected for program reduction. This reduction produces two child nodes, one of which "dies" in the next step because there are no matching program nodes.

(2) The reduction (1) partially instantiates the parse tree Tree. The literal wf(C/[T1,T2]) is selected for program reduction.

(3) This produces a clause body that contains literals that refer to the subtree T1 and literals that refer to the subtree T2. The computation rule in effect partitions the literals and selects those that refer to the subtree T1 (because they are associated with an instantiated left string argument).

[1] When used with a program encoding a context-free grammars in the manner of Figure 1, the Lemma Table proof procedure with the control rule R simulates Earley's CFG parsing algorithm [3]. The operations in the Lemma Table proof procedure are named after the corresponding operations of Earley's algorithm.

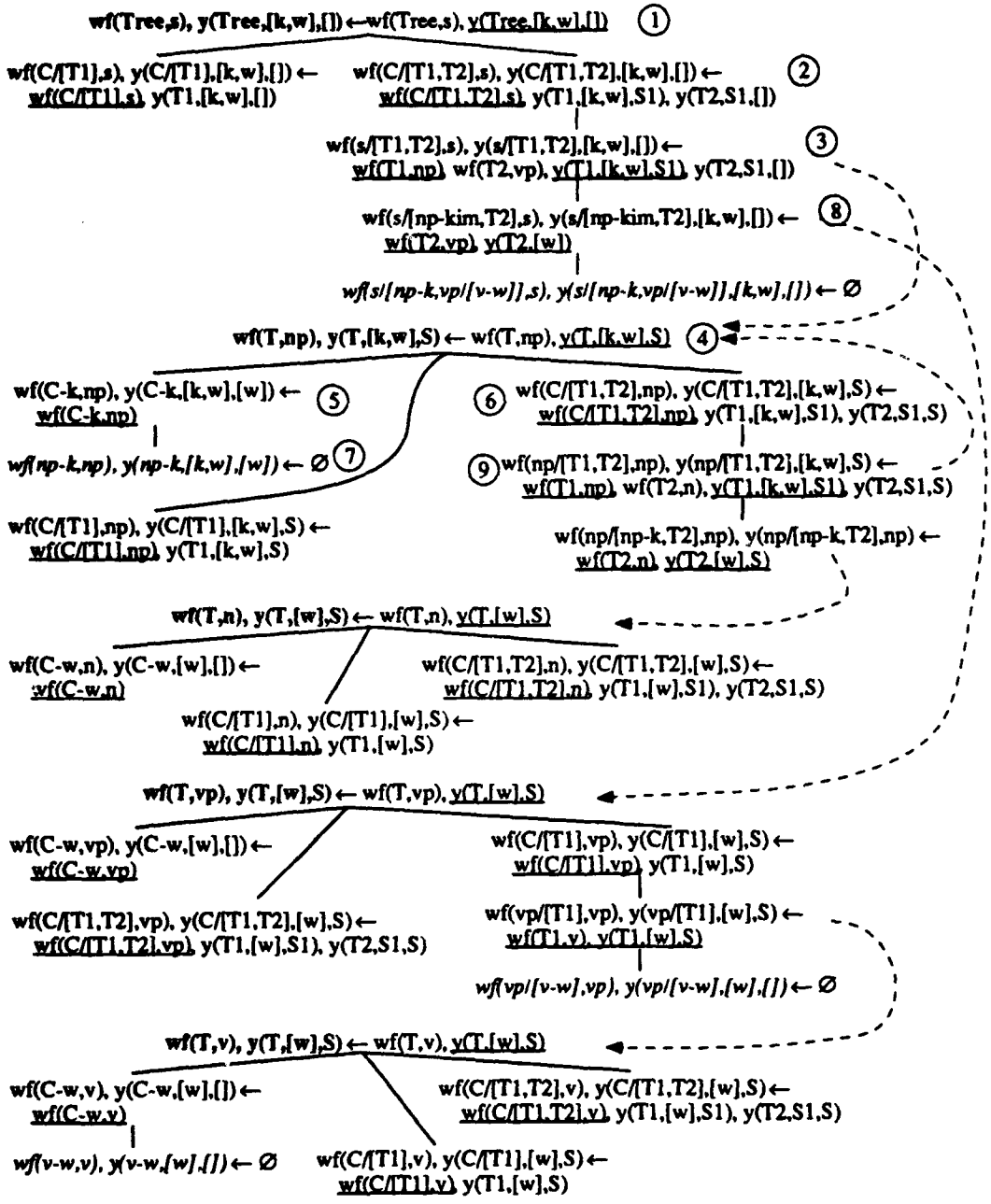


Figure 3: A lemma table for wf(Tree,s), y(Tree,[kim,walks],[])

(4) A new table entry is created for the literals selected in (3). Informally, this entry searches for an NP located at the beginning of the utterance. Because this node is a root node, the literal y(T,[kim,walks],S) is selected for program expansion.

(5–6) The literals with predicate *wf* are selected for program expansion.

(7) The body of this node's clause label is empty, so it's label is added to the solutions list of the table entry. Informally, this node corresponds to the string [kim] having been recognized as an NP.

(8) The solution found in (7) is incorporated into the tree beneath (3). A new table entry is generated to search for a VP spanning the string [walks].

(9) Just as in (3), the literals in the body of this clause's label refer to two distinct subtrees, and as before the computation rule selects the literals that refer to T1. However there is already a table entry (4) for the selected goal, so a new table entry is not created. The solutions already found for (4) generate a child node to search for an N beginning at walks. No solutions are found for this search.

4 Soundness and Completeness

This section demonstrates the soundness and completness of the lemma table proof procedure. Soundness is straight-forward, but completeness is more complex to prove. The completeness proof relies on the notion of an unfolding of a lemma tree, in which the table nodes of a lemma tree are systematically replaced with the tree that they point to. In the limit, the resulting tree can be viewed a kind of SLD proof tree, and completeness follows from the completeness of SLD resolution.

Theorem 1 (Soundness) *If the output of lemma table proof procedure contains a clause* $G \leftarrow C$, *then* $P \models C \rightarrow G$.

Proof: Each of the clause labels on lemma tree nodes is either a tautology or derived by resolving other clause labels and program clauses. Soundness follows by induction on the number of steps taken by the proof procedure. ■

As might be expected, the completeness proof is much longer than the soundness proof. For space reasons it is only sketched here.

For the completeness proof we assume that the control rule R is such that the output γ of the lemma table proof procedure contains only clauses with empty bodies. This is reasonable in the current context, because non-empty clause bodies correspond to goals that have not been completely reduced.

Then completeness follows if for all P, G and σ, if $P \models G\sigma$ then there is an instance G' on the solution list that subsumes $G\sigma$.

Further, without loss of generality the abstraction operation α is assumed to be the identity function on goals, since if $\alpha(G(\vec{t})) = G(\vec{t'})$, the goal $G(\vec{t})$ can be replaced with the equivalent $G(\vec{t'}) \cup \{\vec{t'} = \vec{t}\}$ and α taken to be the identity function.

Now, it is a corollary of the Switching Lemma [11, pages 45–47] that if $P \models G\sigma$ then there is an n such that for any computation rule there is an SLD refutation of G of length n whose computed answer substitution θ subsumes σ. We show that if θ is a computed answer substitution for an SLD derivation of length n then there is a node tagged solution and labelled $G\theta \leftarrow \emptyset$ in lemma tree T for the top-level goal.

First, a well-formedness condition on lemma trees is introduced. Every lemma tree in a lemma table at the termination of the lemma table proof procedure is well-formed. Well-formedness and the set of nodes tagged solution are preserved under an abstract operation on lemma trees called expansion. The expansion of a lemma tree is the tree obtained by replacing each node tagged table(B',p) with the lemma tree in the table entry pointed to by p.

Because expansions preserves well-formedness, the lemma tree T' resulting from n iterated expansions of the lemma tree T for the top-level goal is also well-formed. Moreover, since the root node of every lemma tree is required to be tagged program, all nodes in T' within distance n arcs of the root will be tagged program or solution. This top part of T' is isomorphic to the top part of an SLD tree T_s for G, so if θ is a computed answer substitution for an SLD derivation in T_s of length n or less then there is a node tagged solution and labelled $G\theta \leftarrow \emptyset$ in T', and hence T. Since n was arbitrary, every SLD refutation in T_s has a corresponding node tagged solution in T' and hence in T.

5 Conclusion

This paper generalizes standard memoization techniques for logic programming to allow them to be used for constraint logic programming. The basic informational unit used in the Lemma Table proof procedure is the generalized clause $G \leftarrow C$. Generalized clauses can be given a constraint interpretation as "any interpretation which satisfies the constraints C also satisfies G". The lemmas recorded in the lemma table state how sets of literals are reduced to other sets of literals. Because the heads of the lemmas consist of sets of literals rather than just individual literals, the lemmas express properties of systems of constraints rather than just individual constraints. Because the solutions recorded in the lemma table can contain unresolved constraints, it is possible to pass constraints out of a lemma into the superordinate computation.

In this paper G and C were taken to be sets of literals and the constraints C were defined by Horn clauses. In a more general setting, both G and C would be permitted to contain constraints drawn from a specialized constraint language not defined by a Horn clause program. Höhfeld and Smolka [5] show how to extend SLD resolution to allow general constraints over arbitrary domains. Their elegant relational approach seems to be straight-forwardly applicable to the Lemma Table proof procedure, and would actually simplify its theoretical description because equality (and unification) would be treated in the constraint system. Unification failure would then be a special case of constraint unsatisfiability, and would be handled by the "optimization" described by Höfeld and Smolka that permits nodes labelled with clauses $G \leftarrow C$ to be deleted if C is unsatisfiable.

References

[1] Chen, W. and D.S. Warren. 1989. *C-Logic of Complex Objects.* ms. Department of Computer Science, State University of New York at Stony Brook.

[2] Chomsky, N. 1986. *Knowledge of Language: Its nature, origins and use.* Praeger. New York.

[3] Earley, J. 1970. "An efficient context-free parsing algorithm", in *Comm. ACM* 13:2, pages 94–102.

[4] Giannesini, F., H. Kanoiui, R. Pasero and M. van Canegham. 1986. *Prolog.* Addison-Wesley. Reading, Massachusetts.

[5] Höhfeld, M. and G. Smolka. 1988. *Definite Relations over Constraint Languages* Lilog report 53, IBM Deutschland.

[6] Johnson, M. 1989. "The Use of Knowledge of Language", in *Journal of Psycholinguistic Research*, 18.1.

[7] Johnson, M. 1990. "Features, frames and quantifier-free formulae", in P. Saint-Dizier and S. Szpakowicz, eds., *Logic and Logic Grammars for Language Processing*, Ellis Horwood, New York, pages 94–107.

[8] Johnson, M. 1991. "Deductive Parsing: The Use of Knowledge of Language", in R.C. Berwick, S.P. Abney and C. Tenny, eds., *Principle-based Parsing: Computational and Psycholinguistics*, Kluwer Academic Publishers, Dordrecht, pages 39–65.

[9] Johnson, M. 1991. "Techniques for deductive parsing", in C.G. Brown and G. Koch, eds., *Natural Language and Logic Programming III*, North Holland, Amsterdam, pages 27–42.

[10] Johnson, M. 1992. "The left-corner program transformation", ms., Brown University.

[11] Lloyd, J. 1984. *Foundations of Logic Programming.* Springer-Verlag, Berlin.

[12] Matsumoto, M., H. Tanaka, H. Hirakawa, H. Miyoshi and H. Yasukawa. 1983. "BUP: a bottom-up parser embedded in Prolog", in *New Generation Computing* 1:2, pages 145–158.

[13] Pereira, F. and S. Shieber. 1987 *Prolog and Natural Language Analysis.* CSLI Lecture Notes Series, Chicago University Press.

[14] Pereira, F. and D.H. Warren. 1983. "Parsing as Deduction", in *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics.* MIT, Cambridge, Mass.

[15] Shieber, S. 1985. *Introduction to Unification-based theories of Grammar.* CLSI Lecture Notes Series, Chicago University Press.

[16] Smolka, G. 1992. "Feature Constraint Logics for Unification Grammars", in *The Journal of Logic Programming* 12:1-2, pages 51–87.

[17] Tamaki, H. and T. Sato. 1986. "OLDT resolution with tabulation", in *Proceedings of Third International Conference on Logic Programming*, Springer-Verlag, Berlin, pages 84–98.

[18] Vieille, L. 1987. "Database-complete proof procedures based on SLD resolution", in *Logic Programming: Proceedings of the fourth international conference*, The MIT Press. Cambridge, Massachusetts.

[19] Vieille, L. 1989. "Recursive query processing: the power of logic", Theoretical Computer Science 69, pages 1–53.

[20] Warren, D. S. 1992. "Memoing for logic programs", in *Communications of the ACM* 35:3, pages 94–111.

Local Consistency in Parallel Constraint-Satisfaction Networks

Simon Kasif
Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218
kasif@cs.jhu.edu

Arthur L. Delcher
Computer Science Department
Loyola College in Maryland
Baltimore, MD 21210
delcher@loyola.edu

Abstract

We summarise our work on the parallel complexity of local consistency in constraint networks, and present several basic techniques for achieving parallel execution of constraint networks. We are interested primarily in developing a classification of constraint networks according to whether they admit massively parallel execution. The major result supported by our investigations is that the parallel complexity of constraint networks is critically dependent on subtle properties of the network that do not influence its sequential complexity.

1 Introduction

In this position paper we summarize our work on the parallel complexity of local consistency in constraint networks [Kas90, Kas86, Kas89, KRS87, KD90]. Our research is aimed at deriving a precise characterization of the utility of parallelism in such networks. We are interested primarily in developing a classification of constraint networks according to whether they admit massively parallel execution. We have analyzed parallel execution for chain networks, tree networks, two-label networks, directed-support networks and path consistency in general networks. For example, we show that contrary to common intuition, chain networks do admit fast parallel solutions (as in fact do all acyclic graphs). While the obvious parallel algorithm for local consistency in constraint networks should work well in practice, we would like to obtain lower and upper bounds on the complexity of the problem on ideal parallel machines (such as the PRAM).

This study may have significant practical implications since it should indicate which parallel primitives are fundamental in the solutions of large constraint systems. Once such primitives are implemented in hardware, they execute in essentially constant time for all practical purposes, *e.g.*, parallel prefix on the Connection Machine. The original design of the Connection Machine was motivated by these considerations. The ultimate goal of our research is to produce a set of primitives that are critical to the solution of constraint problems.

2 Constraint Satisfaction, Local Consistency and Discrete Relaxation

Constraint satisfaction networks are used extensively in many AI applications such as planning, scheduling, natural language analysis, truth-maintenance systems, and logic programming [dK86, HS79, Mac77, RHZ76, Win84, VH89a]. These networks use the principle of local constraint propagation to achieve global consistency (*e.g.*, consistent labelling in vision).

A constraint satisfaction network can be defined as follows. Let $V = \{X_1, \ldots, X_n\}$ be a set of variables. With each variable X_i we associate a set of labels L_i. Now let $\{R_{ij}\}$ be a set of binary predicates that define the compatibility of assigning labels to pairs of variables. Specifically, $R_{ij}(a, b) = 1$ iff the assignment of label a to X_i is compatible with the assignment of label b to X_j.

The Constraint Satisfaction Problem (CSP) is defined as the problem of finding an assignment of labels to the variables that does not violate the constraints given by $\{R_{ij}\}$. More formally, a solution to CSP is a

vector $(a_1, \ldots, a_n)$ such that a_i is in L_i and for each i and j, $R_{ij}(a_i, a_j) = 1$.

A standard approach to model CSP problems is by means of a constraint graph, (*e.g.*, *v.* [Mac77, Mon74]). The nodes of the constraint graph correspond to variables of the constraint network, and the edges correspond to the binary constraints. In this context, each edge in the constraint graph is labelled with a matrix that shows which assignments of labels to the objects connected by that edge are permitted. In this interpretation CSP can be viewed as generalized graph coloring.

Since CSP is known to be $\mathcal{NP}$-complete, several local-consistency algorithms have been used extensively to filter out impossible assignments.

Arc consistency allows an assignment of a label a to an object X iff for every other object X' in the domain there exists a valid assignment of a label a' which does not violate the constraints [Mac77, Mon74]. More formally, we define arc consistency as follows.

Given a constraint network, a solution to the *local version of CSP* or *arc consistency* (AC) is a vector of sets $(M_1, \ldots, M_n)$ such that M_i is a subset of L_i and label a is in M_i iff for every M_j, $i \neq j$ there is a label b in M_j, such that $R_{ij}(a, b) = 1$. Intuitively, a label a is assigned to a variable iff for every other variable there is at least one valid assignment of a label to that other variable that supports the assignment of label a to the first variable.

We call a solution $(M_1, \ldots, M_n)$ a *maximal* solution for AC iff there does not exist any other solution $(S_1, \ldots, S_n)$ such that $M_i \subseteq S_i$ for all $1 \leq i \leq n$. We are interested only in maximal solutions for an AC problem. By insisting on maximality we guarantee that we are not losing any possible solutions for the original CSP. Therefore, in the remainder of this paper a solution for an AC problem is identified with a maximal solution. The sequential time complexity of AC is discussed in [MF85, MH86, Kas90]. Discrete relaxation is the most commonly used method to achieve local consistency. Starting with all possible label assignments for each variable, *discrete relaxation* repeatedly discards labels from variables if the AC condition specified above does not hold.

3 Parallel Processing of Constraint Networks

The standard approach for achieving arc consistency is the following discrete-relaxation procedure:

Procedure Parallel-AC

Step 1 Start by setting $M_i = L_i$, for $1 \leq i \leq n$.

Step 2 Repeat the following:

> For each constraint between X_i and X_j test whether for each label $a \in M_i$ there exists a label $b \in M_j$ that permits it. If there is no such b then remove a from M_i.

until no label is removed from any M_i.

It is easy to see that this algorithm will terminate in $O(EK^2nK)$ time, where E is the number of edges in the constraint-network graph and K is the number of labels for each variable, (recall that EK^2 is the size of the input). In fact, much better sequential algorithms for the problem are discussed in [MF85, MH86, Kas89, Kas90].

Clearly procedure Parallel-AC can be parallelized in a straightforward way. If we assume a CRCW PRAM as our model of parallel computation,[1] we have the following simple result:

Claim: The parallel complexity of procedure Parallel-AC is $O(nK)$ on a CRCW PRAM with EK^2 processors.

Proof: Simply assign K processors to each arc and label, and perform the test for arc consistency in Step 2 in parallel. Arc consistency is essentially a logical OR on set membership of a set of labels, and can be performed in constant time on a CRCW PRAM. At each parallel step, if the algorithm does not halt, then at least one label must be dropped and there are a total of nK labels. □

[1] CRCW PRAM is a standard shared-memory parallel computation model that permits concurrent reads and writes into the same location. Concurrent writes are permitted if they agree on the data being written.

On a more realistic model of computation, such as an EREW (exclusive read/exclusive write) PRAM, we can perform the above procedure in $O(nK \log K)$ parallel time, the extra $\log K$ factor being required to compute the logical OR.

It is easy to see that the procedure above has a lower bound of nK steps, *i.e.*, in the worst case it does not fully parallelize in the sense of achieving polylogarithmic parallel time. As a simple example, consider a chain constraint graph. In [Kas90] we proved the following much stronger result, namely, that AC is inherently sequential in the worst case for general constraint networks.

Theorem 1: (Kasif 90) The propositional Horn clause satisfiability problem is log-space reducible to the AC problem. That is, AC is $\mathcal{P}$-complete.

Local consistency belongs to the class of inherently sequential problems called log-space complete for $\mathcal{P}$ (or $\mathcal{P}$-complete). Intuitively, a problem is $\mathcal{P}$-complete iff a polylogarithmic-time parallel solution (with a polynomial number of processors) for the problem will produce a polylogarithmic-time parallel solution for every deterministic polynomial-time sequential algorithm. This implies that unless $\mathcal{P} = \mathcal{NC}$ ($\mathcal{NC}$ is the class of problems solvable in polylogarithmic parallel time with polynomially many processors) we cannot solve the problem in polylogarithmic time using a polynomial number of processors. The above theorem implies that to achieve polylogarithmic parallel time one (probably) would need a superpolynomial number of processors. We emphasize that this is a worst-case result and indeed several groups have reported successful experiments with massively parallel constraint processing [DdK88, SH87, VH89b].

We first make an observation which is critical to the understanding of the procedural semantics, and consequently the complexity, of achieving arc consistency. In [Kas86, Kas89, Kas90] we provided a two-way reduction between arc consistency and Propositional Horn Satisfiability (PHS). Specifically, given a constraint satisfaction problem S one can construct a propositional Horn formula (AND/OR graph) G such that arc consistency of S can be achieved by (essentially) running a satisfiability algorithm for G. For a formal construction see [Kas90]. We sketch the intuition here. For each label a and a variable X we construct a propositional atom $P_{X,a}$ which means a drops from X. We also use a propositional atom $Q_{X,a,Y}$ to mean that variable Y has no label that supports label a in X. Consider, for example, a variable X connected in the constraint graph to variables Y and Z. Assume that Y has labels a_1, a_2 and Z has labels a_3, a_4 that support a at X. Thus, we construct the formulae:

$$
\begin{aligned}
P_{X,a} &\leftarrow Q_{X,a,Y} \\
P_{X,a} &\leftarrow Q_{X,a,Z} \\
Q_{X,a,Y} &\leftarrow P_{Y,a_1} \wedge P_{Y,a_2} \\
Q_{X,a,Z} &\leftarrow P_{Z,a_3} \wedge P_{Z,a_4}
\end{aligned}
$$

We can apply one iteration of procedure Parallel-AC to determine which labels will be dropped initially.

Note that if the constraint graph has E edges and K labels per variable, the size of the formulae is potentially EK^2. More importantly, the sequential complexity of solving satisfiability of this graph is $O(EK^2)$ (see [Kas90] for details). This is a slight improvement over the result in [MF85] and it matches the algorithm in [MH86]. The advantage is that we can use this reduction to derive optimal algorithms for AC when the resulting Horn-clause formula is of small size. Thus, for example, when the size of the formula is $O(EK)$ we get an $O(EK)$ algorithm. We also can devise efficient parallel algorithms when the resulting Horn-clause formula has some special graph structure. Note that this reduction is from AC to PHS, as opposed to the reduction from PHS to AC used to prove Theorem 1.

4 AC in Dense Graphs

It has been noted by several researchers that for many P-complete problems such as depth-first search, circuit evaluation and unification, optimal speed-up is possible if the underlying graph is very dense, *i.e.*, the number of edges in the graph is quadratic [VS86]. We illustrate this simple principle with an example. Assume we are given a boolean circuit that consists of NAND gates only. The circuit contains N gates and E edges. We also are given an input to the circuit. Clearly, any sequential algorithm must take $O(E)$ time to evaluate this circuit. The standard technique for evaluating circuits uses a counter for each gate. The initial value of the counter is set to the number of inputs the gate has. We maintain a queue of gates whose

value has been computed (initially, just the input gates). We pick any gate on the queue and traverse all its outgoing edges. For each such edge we decrement the counter associated with the gate incident to the edge and drop the edge. If the counter becomes zero we simply add the gate to the queue. Note that this simple algorithm is in fact the same algorithm that yields optimal sequential time for both PHS and AC [MH86]. PHS and AC problems may generate circuits with cycles but this does not fundamentally change the algorithm, as was pointed out in [Kas90].

But now consider a brute-force parallel algorithm, where we essentially perform the sequential algorithm, with one exception. When we choose a gate g from the queue we use N processors to update the counters of all the gates that are connected to g. This can be trivially done in constant time on a PRAM (parallel shared-memory machine). When any counter becomes zero, we add the gate to the queue as in the sequential version (on some models of parallel computation the above two steps may take logarithmic time). Since we are visiting at most N nodes the procedure terminates in $O(N)$ steps using N processors. Thus we achieve linear speed-up when the number of edges in the circuit is $O(N^2)$. This observation holds for Propositional Horn Satisfiability. It also holds for Arc Consistency and Path Consistency by using the reduction to Propositional Horn Satisfiability (or circuit evaluation). For AC, this observation yields $O(nK)$ performance with nK processors (details are left as an exercise to the reader).

Theorem 2: AC can be solved in $O(nK)$ time with nK processors.

The algorithm above will work well on any machine that can support broadcasts (to implement traversing edges) and some kind of fast selection operation (to pick the next node from the queue). Parallel prefix on the connection machine can support both primitives efficiently in logarithmic time. Note, that the parallel algorithm described above can be improved in practice by retrieving (in parallel) all nodes (gates) from the queue. Thus, for a tree-structured graph we will achieve performance proportional to the depth rather than the size of the tree.

5 Summary of Parallel AC Results

In this section we summarize our knowledge of the parallel complexity of computing local consistency in constraint satisfaction problems. The results appear in Tables 1 and 2. We have classified the parallel complexity of problems into two classes: $\mathcal{P}$-complete problems and $\mathcal{NC}$ problems. $\mathcal{P}$-complete problems are perceived to be difficult to parallelize (in the same sense that $\mathcal{NP}$-complete problems are considered intractable), and $\mathcal{NC}$ problems can be solved in polylogarithmic time with a polynomial number of processors. $\mathcal{NC}$ problems are often amenable for optimal speed-up on parallel machines. In both tables R denotes the binary compatibility predicate.

The main practical conclusions that we can draw from our study are as follows:

1. Local consistency in constraint networks is generally $\mathcal{P}$-complete. Practical experience suggests that it is difficult to obtain optimal parallel algorithms for such problems. By optimal parallel algorithms we mean algorithms that obtain P-fold speed-up of the best sequential algorithm with P processors. However, it is often easy to obtain optimal speed-ups for these (and other) problems when the number of processors is small.

2. Substantial speed-ups for parallel local consistency algorithms are possible if the constraint graph is dense. This can be accomplished using a simple obvious algorithm which is likely to be efficient in practice. Specifically, given a constraint network with n nodes and K labels per node, it is easy to obtain an $O(nK)$ algorithm for arc-consistency with nK processors on a shared-memory parallel model of computation.

3. The application of the obvious parallel version of the arc-consistency algorithm to such networks does not yield a sublinear algorithm. The parallel complexity of local consistency in chain networks has been shown equivalent to reachability problems in directed graphs. While this class of networks can theoretically be solved very fast, in practice our results imply that the "transitive closure" bottleneck may apply to chain networks. It currently is not known how to get optimal speed-ups for transitive closure problems in graphs unless the number of processors is smaller than the number of nodes in the graph.

Table 1: Complexity of Arc Consistency for Arbitrary-Size Label Sets

Arbitrary K (K is the size of the label set L)		
G	CSP	AC
Chain	$\mathcal{NC}$; reachability	Undirected R's: $\mathcal{NC}$; reachability Directed R's: P-complete; reduction from Propositional Horn-Clause Solvability
Tree	$\mathcal{NC}$	Undirected R's: $\mathcal{NC}$; like expression eval where operation at each node is intersection of sets of support for each label. Directed R's: P-complete; from above
Simple Cycle	$\mathcal{NC}$; reachability	Undirected R's: $\mathcal{NC}$; cycle detection Directed R's: P-complete; from above
Arbitrary Graph	$\mathcal{NP}$-complete; reduction from graph colouring	Undirected R's: P-complete; reduction from Propositional Horn-Clause Solvability Directed R's: P-complete; from above

Table 2: Complexity of Arc Consistency for Fixed-Size Label Sets

Fixed K (K is the size of the label set L)		
G	CSP	AC
Chain	$\mathcal{NC}$	Undirected R's: $\mathcal{NC}$ Directed R's: $\mathcal{NC}$
Tree	$\mathcal{NC}$	Undirected R's: $\mathcal{NC}$ Directed R's: $\mathcal{NC}$; from above
Simple Cycle	$\mathcal{NC}$	Undirected R's: $\mathcal{NC}$ Directed R's: $\mathcal{NC}$; from above
Arbitrary Graph	$K = 2$: Linear sequential algorithm by reduction to 2-SAT which is $\mathcal{NC}$ $K \geq 3$: $\mathcal{NP}$-complete; reduction from 3-colouring graphs	Undirected R's: For $K = 2$, $\mathcal{NC}$ by reachability along "singleton paths"; For $K \geq 3$, P-complete from Propositional Horn-Clause Solvability Directed R's: P-complete for $K \geq 2$

4. For tree networks we suggested several algorithms that achieve sublinear time with many parallel processors. We have been unable to find an obvious practical algorithm to achieve optimal speed-up in tree networks.

5. We provided a reduction from i-consistency to propositional Horn satisfiability which allows us to derive optimal sequential algorithms for problems such as path consistency in a simple manner.

Acknowledgements This research has been supported partially by the Air Force Office of Scientific Research under grant AFOSR-89-1151 and the National Science Foundation under grant IRI-88-09324. Thanks are due to David McAllester, Judea Pearl and Rina Dechter for their constructive comments.

References

[DdK88] M. Dixon and J. de Kleer. Massively parallel assumption-based truth maintenance. In *Proceedings of AAAI-88*, pages 199–204, 1988.

[dK86] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[HS79] R. M. Haralick and L. G. Shapiro. The consistent labeling problem: Part I. *IEEE Trans. Patt. Anal. Mach. Intel.*, PAMI-1:173–184, 1979.

[Kas86] S. Kasif. On the parallel complexity of some constraint satisfaction problems. In *Proceedings of the 1986 National Conference on Artificial Intelligence*, August 1986.

[Kas89] S. Kasif. Parallel solutions to constraint satisfaction problems. In *Principles of Knowledge Representation and Reasoning*, May 1989.

[Kas90] S. Kasif. On the parallel complexity of discrete relaxation in constraint networks. *Artificial Intelligence*, pages 229–241, 1990.

[KD90] S. Kasif and A. Delcher. Analysis of local consistency in parallel constraint networks. *Artificial Intelligence (to appear), also in 1991 AAAI Symposium on Constraint Based Reasoning, pp. 154-163*, 1990.

[KRS87] S. Kasif, J. Reif, and D. Sherlekar. Formula dissection: A parallel algorithm for constraint satisfaction. In *Proceedings of the 1987 IEEE Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, October 1987.

[Mac77] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[MF85] A. K. Mackworth and E. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction. *Artificial Intelligence*, 25:65–74, 1985.

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Mon74] U. Montanari. Networks of constraints: Fundamental properties. *Inform. Sci.*, 7:727–732, 1974.

[RHZ76] A. Rosenfeld, R. Hummel, and S. Zucker. Scene labeling by relaxation operations. *IEEE Trans. Syst. Man Cybern.*, SMC-6:420–433, 1976.

[SH87] A. Samal and T.C. Henderson. Parallel consistent labelling algorithms. *International Journal of Parallel Programming*, 16:341–364, 1987.

[VH89a] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[VH89b] P. Van Hentenryck. Parallel constraint satisfaction in logic programming. In *Proceedings of the International Conference on Logic Programming*, 1989.

[VS86] J. S. Vitter and R. A. Simons. New classes for parallel complexity: A study of unification and other complete problems in $\mathcal{P}$. *IEEE Transactions on Computers*, C-35(5):403–418, 1986.

[Win84] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, 1984.

Exploiting Constraint Dependency Information For Debugging and Explanation

Walid T. Keirouz, Glenn A. Kramer, Jahir Pabon

Schlumberger Laboratory for Computer Science
8311 North RR 620
PO Box 200015
Austin, TX 78720-0015
{walid,gak,jahir}@austin.slcs.slb.com

Abstract

Constraint programming is another form of programming and, as such, should be supported by appropriate environments that provide debugging, explanation and optimization capabilities. We are building tools needed for such an environment and using them in the context of geometric constraint programming for graphics and mechanical design. In this paper, we present the components of such an environment and their capabilities. We describe the use of constraint dependency graphs for debugging and explanation, and present an algorithm for identifying constraints that cause a model to be over-constrained.

1 Introduction

Constraints between entities in a model form a declarative specification that is used by a constraint solver to satisfy and maintain relations between entities in the model. The solution steps taken by a solver to satisfy these constraints can be collected in a "solution plan" and can be viewed as a procedural program for solving these constraints. Thus, a constraint model forms a declarative specification of a procedural program that is generated automatically to solve the model's constraints. As such, constraint programming is another form of programming and should be supported by appropriate environments that support debugging, explanation and optimization. We are building tools needed in such an environment and using them in the context of geometric constraint programming for graphics and mechanical design.

Graphics was one of the first domains to which constraint-based techniques were applied [Sutherland, 1963; Borning, 1979]. Since then, commercial constraint-based computer-aided design (CAD) systems have emerged [Mills, 1992]. While they are considered far superior to previous CAD tools, engineers experience difficulty understanding the constraint models that underlie their designs (constraints may be specified by users or inferred by CAD systems). At times, the constraints in a large model become so confusing that a designer might scrap a design and start anew [Tee, 1992]. Furthermore, when a designer returns to the same design several months later, there is no record of how the constraints were solved and what the design dependencies are; the designer must rediscover the original design intent.

People typically develop their own mental models of the plan of how constraints are satisfied by a constraint system. However, these models often do not reflect the true solution plan. The difficulties users encounter are then compounded because there is no way to view and understand the dependencies in the constraint models they have built.

2 Geometric Constraint Engine

We have developed a Geometric Constraint Engine (GCE) that is currently used in a sketching product [Belleville, 1992]. GCE is based on research in the use of geometric constraints in kinematics and conceptual

Constraint name	Explanation
dist:point-point(G_1, G_2, d)	Distance between point G_1 and point G_2 is d.
dist:point-line(G_{pt}, G_ℓ, d)	Distance between point G_{pt} and line G_ℓ is d.
dist:point-plane(G_{pt}, G_{pl}, d)	Distance between point G_{pt} and plane G_{pl} is d.
dist:line-circle(G_ℓ, G_c, d)	Distance between line G_ℓ and circle G_c is d.*
angle:vec-vec(G_1, G_2, α)	Angle between vector G_1 and vector G_2 is α.

* In two dimensions, $d = 0$ represents a tangency constraint.

Table 1: Constraints used in GCE

design [Kramer, 1990; Pabon *et al.*, 1992], and is described in [Kramer, 1992a; Kramer, 1992b]. We are exploring the use of constraint dependency information in the context of GCE.

GCE finds positions, orientations and dimensions of geometric entities in 3D that satisfy a set of constraints relating different entity features. Geometric entities can be nested hierarchically in a part-whole relationship; *aggregate* entities are composed of combinations of *primitive* ones—points, vectors and dimensions.

With the exception of dimensional constraints, all constraints used in GCE are binary constraints—they relate two geometric entities. These constraints may additionally involve real parameters. Examples of constraints used in GCE are shown in Table 1. Dimensional constraints are unary; they relate one geometric entity to a real-valued dimension parameter. Constraints may apply to subparts of a given entity. For example, to constrain two lines to be parallel, one constrains the vectors of those lines to have an angle of zero.

GCE addresses an issue currently outside the major focus of constraint-based systems research: solving highly nonlinear constraint problems over the domain of real numbers.[1] To solve these problems, GCE imposes an operational semantics for constraint satisfaction in the geometry domain. It does so by employing a metaphor of *incremental assembly;* geometric entities are moved to satisfy constraints in an incremental manner. The assembly process is virtual, as geometric entities are treated as *ghost* objects that can pass through each other during assembly. Such an assumption is allowed because the goal of the constraint satisfaction process is to determine globally-consistent locations of the geometric entities rather than the paths required for a physical assembly of that geometry.

GCE assembles geometric entities incrementally to satisfy the constraints acting on them. As the objects are assembled, their degrees of freedom are consumed by the constraints, and geometric invariants are imposed. An operational semantics is imposed: *measurements* and *actions* are used to satisfy each individual constraint. GCE uses information about an entity's degrees of freedom to decide which constraint to solve and to ensure that an action being applied to a geometric entity does not invalidate any geometric invariants imposed by previously-satisfied constraints. This ensures that the solution algorithm is confluent.

The solution algorithms in GCE can handle fully- as well as under- and over-constrained models. The solution of a set of constraints can be captured as a plan that may be replayed to satisfy the constraints when one or more numerical constraint parameters are changed.[2]

3 States of constraint models

A constraint model can be in one or more of several states, which are enumerated here and discussed subsequently. The following notation will be used: $\mathbf{ls}_i$ denotes line segment i; $\mathbf{l}_i\mathbf{p}_1$ denotes end-point 1 of $\mathbf{ls}_i$; $\mathbf{l}_i\mathbf{p}_2$ denotes end-point 2 of $\mathbf{ls}_i$; $\mathbf{v}_i$ denotes the direction vector of $\mathbf{ls}_i$. In the examples in Figure 1, a tick mark in the center of a line segment denotes a fixed dimension constraint for the line segment, an arc with label α_{ij} denotes and angle constraint between the vectors of $\mathbf{ls}_i$ and $\mathbf{ls}_j$, and coincident end-points in the

[1] In the special volume of *Artificial Intelligence* concerned with constraint-based reasoning, seven of the eleven articles address the finite constraint satisfaction problem [AIJ, 1992]. Three of the remaining four address problems using a (linear) interval representation, while [Kramer. 1992b] addresses nonlinear real-valued CSPs.

[2] Note that some parameter changes can alter the topology of the constraint problem and hence require finding another plan. For example, in a **dist:point-point** constraint, changing the distance parameter from zero to non-zero alters the degrees of freedom removed by the constraint.

figure indicate a coincidence constraint exists between those end-points. Using this notation, the possible states of a constraint model are now enumerated.

A constraint model is *fully constrained* when there are no remaining degrees of freedom after all constraints have been satisfied, and where no constraint in the system is redundant. An example is shown in Figure 1(a). Here, the length of $\mathbf{ls}_1$ is fixed, and two angles are known. This corresponds to using the "angle-side-angle" formula of elementary geometry to find all parts of a triangle.

A constraint model may be *under-constrained*. Figure 1(b) is similar to Figure 1(a) except that the dimension of $\mathbf{ls}_1$ has been freed. This example describes an infinite family of "similar" triangles, which can be parameterized by fixing the length of any one of the three line segments.

Over-constrained models result from adding more constraints to a fully constrained model (or by adding a constraint restricting m degrees of freedom to a model with n remaining degrees of freedom, where $m > n$). Figure 1(c) is similar to Figure 1(a), except that constraint α_{32} has been added. In this case, we have chosen $\alpha_{32} = \pi - (\alpha_{13} + \alpha_{21})$, so the model is numerically consistent. Identifying and correctly solving such cases is important in real-world design (*e.g.*, only one hinge is needed to hold a door on a frame in a constraint-based world; the remaining hinges are mathematically redundant, but are quite useful in the physical world).

Over- and under-constrained situations can coexist in the same constraint model. Figure 1(d) shows such an example. The angles are over-constrained but consistent, as in Figure 1(c), but the lengths are under-constrained as in Figure 1(b).

An over-constrained model can also be *inconsistent*, as shown in Figure 1(e). Here, the three angle constraints do not sum to π. If α_{13} and α_{32} are satisfied, the problem is fully constrained. When α_{21} is asserted, $\mathbf{ls}_2$ would need to be rotated to the dashed position to satisfy the new constraint. However, $\mathbf{ls}_2$ is already fully constrained and hence cannot move to the new position.

Fully constrained systems can be *numerically unsatisfiable*, as shown in Figure 1(f). Here, α_{13} and α_{21} are chosen so that $\mathbf{ls}_3$ and $\mathbf{ls}_2$, both still of indeterminate length, are parallel. Thus, the desired coincidence constraint between $\mathbf{l}_3\mathbf{p}_1$ and $\mathbf{l}_2\mathbf{p}_2$, shown as a dashed line, cannot be satisfied. The number of degrees of freedom removed from the system by the constraints is the same as in Figure 1(a); only the numerical values have changed.

3.1 Reasoning tasks for consistent models

For fully- and under-constrained models, a user may want to replay the solution with different parameter values. Single stepping through the constraint solution can help explain the solution to the user, and allow exploration alternative solutions (by varying the constraint set).

For over-constrained but consistent models, the user may want to compute the sets of over-constraining constraints to determine which constraints should be removed from the model. The user may also want to replay portions of the solution, perhaps in single-step mode, to help explain where the over-constraint lies.

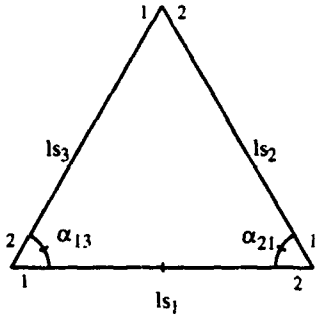
3.2 Reasoning tasks for inconsistent models

There are two modes of failure for constraint models. The first one is when the model is over-constrained and inconsistent because of a conflict between some constraints in the model. The model cannot be made consistent unless some constraints are removed from the model.

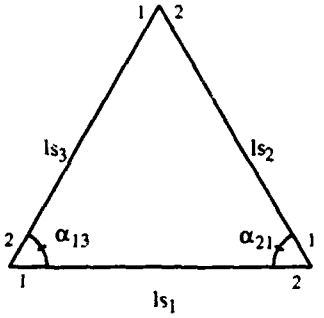
In this case, the model's *over-constraining set* of constraints must be identified, and one constraint from this set must be selected and removed from the model to alleviate the over-constrained situation. The over-constraining set is defined as the set of constraints C such that retraction of any constraint in C will remove the source of over-constraint, and retraction of any constraint not in C has no effect on the over-constraint. This set includes, but is not limited to, the most recently added constraint[3] and the constraints with which it conflicts directly.

The second failure mode is when a model is not over-constrained, but cannot be satisfied due to numerical inconsistencies. In this case, we can identify the unsatisfiable constraints. A plan stepper, which is described below, can then be used to explain where and why the conflict arises.

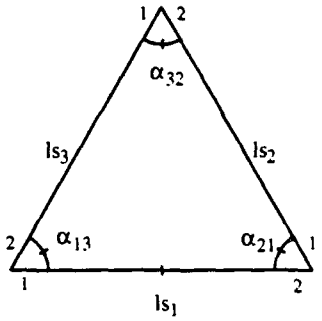
[3] Assuming the constraint system is solved each time a new constraint is added.



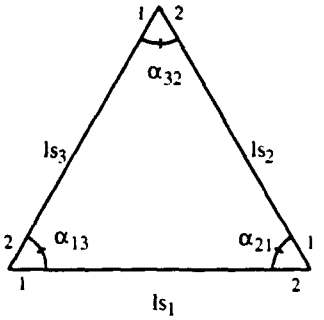
(a) fully constrained



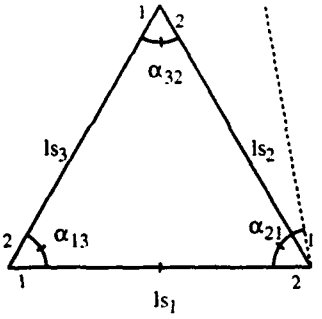
(b) under-constrained



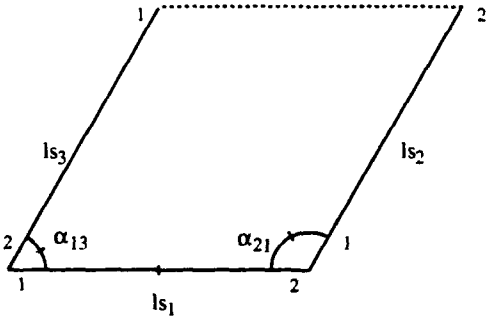
(c) over-constrained (consistent)



(d) over- and under-constrained



(e) over-constrained (inconsistent)



(f) numerically unsatisfiable

Figure 1: Various states of geometric constraint systems

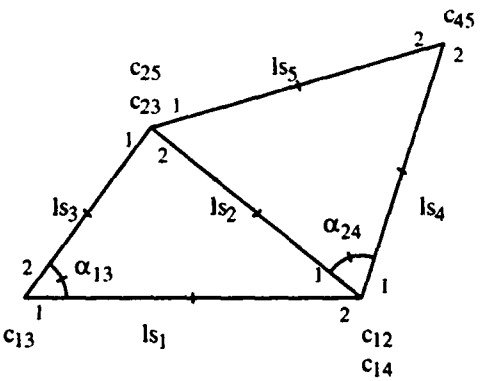


Figure 2: An over-constrained model.

4 Constraint dependency graphs

A constraint system's solution algorithm solves the constraints in a "particular" order that respects the dependencies built into the model and captures this order in a solution plan. However, the captured order is one of many possible orders in which the constraints may be solved. Ordering relationships can be extracted from the solution plan and collected into a dependency graph for the constraint model. The solution plan then describes one of several possible traversals of the nodes in the graph.

Dependency graphs provide a mechanism for constraint dependency analysis which supports debugging and explanation, consistency analysis, user interaction, parameterization of models, and computational optimization.

4.1 Example: an over-constrained model

Figure 2 shows an over-constrained model. The two triangles are fully determined by the coincidence constraints for the line segments' end-points, and by the lengths of the line segments (all of which are fixed). The notation c_{ij} indicates one end-point of $\mathbf{ls}_i$ is coincident with one end-point of $\mathbf{ls}_j$; the end-points are numbered in the diagram. $\mathbf{l}_1\mathbf{p}_1$ and $\mathbf{v}_1$ are fixed in space, which "grounds" the assembly in space. The two angle constraints are then added: these are redundant and lead to over-constraint. While these angle constraints are the direct cause of the over-constraint, removing other constraints could alleviate the problem just as easily.

Figure 3 shows the dependency graph for the model of Figure 2. A special node labeled "S" (for "Start"), is the source node. Square nodes indicate constraints, while circular nodes indicate primitive geometric entities whose locations have become known—or fixed—by having been moved to satisfy the constraint immediately preceding it in the graph.

The constraint nodes labeled "g" depict the unary constraints which ground the location of $\mathbf{l}_1\mathbf{p}_1$ and the orientation of $\mathbf{v}_1$, as well as the dimensions of all the line segments. The c_{ij} and α_{ij} nodes correspond to the constraints in Figure 2. The square nodes marked "I" denote *inference* nodes: Since the representation of line segments is redundant, some information can be inferred after a constraint is satisfied. For example, given $\mathbf{l}_1\mathbf{p}_1$, $\mathbf{v}_1$, and dimension $\mathbf{d}_1$, the location of the other end-point $\mathbf{l}_1\mathbf{p}_2$ may be inferred.

In this graph, the dimensions $\mathbf{d}_2$ and $\mathbf{d}_5$ are found through two different paths (one by grounding, and another by inference from other knowns). These nodes indicate two areas of over-constraint in the model. Each will have its own over-constraining set, which will be derived in Section 5.1.

5 Debugging and explanation

Debugging and explanation involve analyzing the consistency of a constraint model, analyzing the degrees of freedom in the model, and replaying and modifying the order in which the constraints are solved.

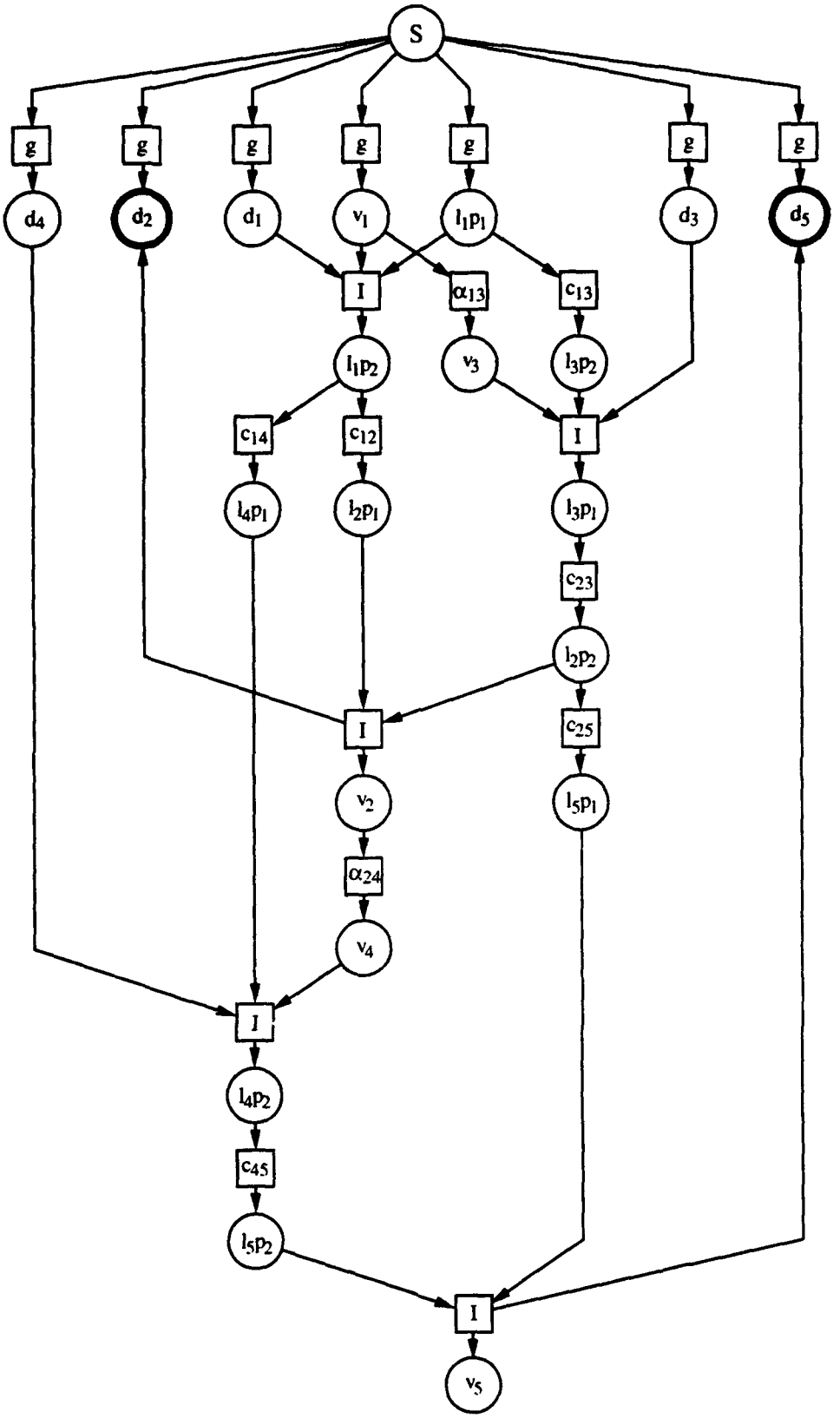


Figure 3: Dependency graph of example of Figure 2.

5.1 Over-constrained models

Although over-constraint is directly caused by the addition of a single constraint to a model, several constraints are involved in the situation. The over-constraining set consists of the constraint that "caused" the over-constraint and a subset of the model's constraints that conflict with that constraint, directly or indirectly. Selecting one constraint from this set and removing it[4] from the model reduces the over-constrained state to a fully- or under-constrained one. We must now identify the over-constraining set; a potentially different set exists for each over-constrained situation within the same model.

Consider the case of the twice-determined dimension $\mathbf{d}_2$. One determination of $\mathbf{d}_2$ comes from the inference node that points to $\mathbf{d}_2$. That inference node states that if $\mathbf{l}_2\mathbf{p}_1$ and $\mathbf{l}_2\mathbf{p}_2$ are both known, then we can determine $\mathbf{d}_2$ and $\mathbf{v}_2$. *If* this inference were not possible, then the source of over-constraint would disappear. We indicate this on the graph of Figure 4 by "flipping" the direction of the arc (originally from "I" to $\mathbf{d}_2$) to be from $\mathbf{d}_2$ to the "I" node. Continuing the reasoning, the inference would not be possible if *either* $\mathbf{l}_2\mathbf{p}_1$ were unknown or $\mathbf{l}_2\mathbf{p}_2$ were unknown. This is indicated by flipping both arcs. Continuing backward from $\mathbf{l}_2\mathbf{p}_1$, this point's location would not be known if $\mathbf{l}_1\mathbf{p}_2$ were unknown, so removing the coincident constraint c_{12} would alleviate the over-constraint.

This chain of reasoning continues back through $\mathbf{d}_1$, but not through $\mathbf{v}_1$ or $\mathbf{l}_1\mathbf{p}_1$. The reason is due to the fanout at those nodes, where information is used in two separate chains of constraint solution, and where this information is recombined by an inference node later on. The concept is analogous to the notion of reconvergent fanout in the digital test generation literature [Breuer and Friedman, 1976]. The algorithm for finding the over-constraining set is then as follows:

1. Beginning with the node at which the over-constraint is detected (*i.e.*, the node which has two arcs leading to it), flip the arcs backward through the next set of nodes.
2. Continue flipping the arcs backward until a node is reached where there is reconvergent fanout (such a node is guaranteed to exist due to the existence of node "S").
3. All constraints for which both input and output arcs are members of the set of flipped arcs are members of the over-constraining set.

The over-constraining sets for the graph of Figure 3 are depicted in Figure 4. The set in long dashes corresponds to the constraints that over-determine $\mathbf{d}_2$. The set in short dashes corresponds to the constraints that over-determine $\mathbf{d}_5$. Note that the reconvergent fanout termination criterion for arc flipping is considered separately for each over-constraining set, *i.e.*, two flipped arcs, one from the $\mathbf{d}_2$ set and one from the $\mathbf{d}_5$ set, do not interact and hence are not reconvergent.

Note that the over-constraint of $\mathbf{d}_5$ has no dependence on $\mathbf{d}_2$, although the overconstraining sets have members in common. One might conclude that, since $\mathbf{ls}_2$ is a part of triangle $\langle \mathbf{ls}_1, \mathbf{ls}_2, \mathbf{ls}_3 \rangle$ (call it **T1**) and that triangle **T1** depends on triangle $\langle \mathbf{ls}_2, \mathbf{ls}_4, \mathbf{ls}_5 \rangle$(call it **T2**), that the over-constraint of **T2** would be dependent on everything in **T1**. However, a closer examination of the particular constraints shows that, even if $\mathbf{d}_2$ were not specified, the location of $\mathbf{l}_2\mathbf{p}_2$ would be found. Also, **T2** is not dependent on $\mathbf{l}_2\mathbf{p}_1$ at all; rather it is dependent on the location of $\mathbf{l}_1\mathbf{p}_2$, to which $\mathbf{l}_2\mathbf{p}_1$ is made coincident.

Some of the constraints identified may remove more degrees of freedom than occur in the the over-constrained situation; removing such a constraint would result in an under-constrained model. For example, the over-determination of $\mathbf{d}_2$ indicates over-constraint of one degree of freedom. Thus, removing an angle constraint like α_{13} removes the over-constraint, but removing a coincidence constraint such as c_{12} removes two degrees of freedom, leading to under-constraint. To avoid this, our tools will suggest altering the constraint to require a non-zero distance between $\mathbf{l}_1\mathbf{p}_2$ and $\mathbf{l}_2\mathbf{p}_1$, which relaxes one degree of freedom. Such suggestions are easy to make automatically.

From a theoretical point of view, all the constraints in an over-constraining set are candidates for removal from a model. However, these sets may need to be filtered depending on the context. For example, in geometric constraint models, constraints that reflect topological relationships (*e.g.*, coincident-points constraints) may have precedence over constraints that specify dimensions.

At present, we have studied the case where the constraints can be solved by pure propagation; we have not yet explored solutions which require the hierarchical structuring utilized by GCE's strategies of loop

[4] Or altering its numerical parameters such that the constraint removes fewer degrees of freedom.

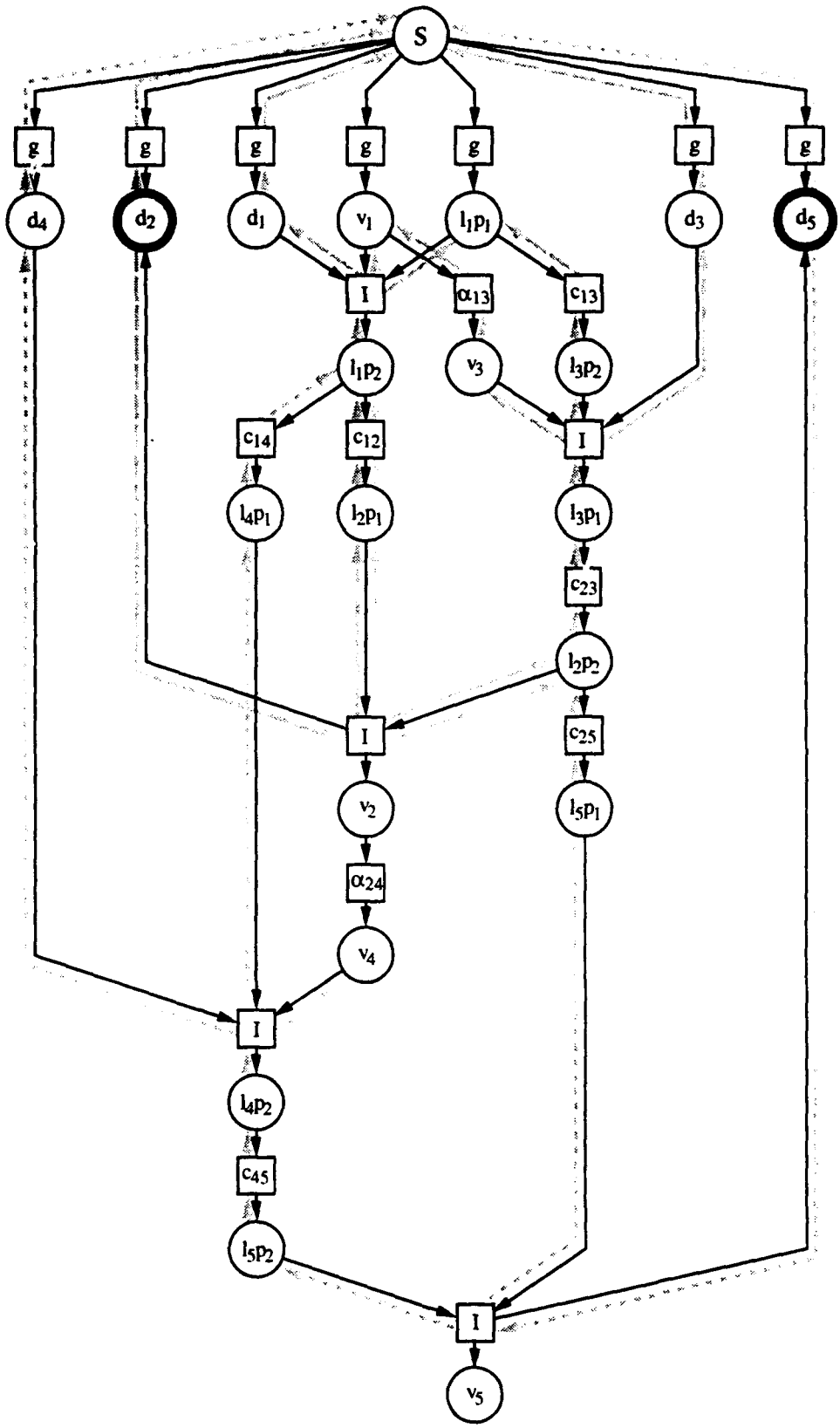


Figure 4: Over-constraining sets for the dependency graph of Figure 3.

analysis and chain analysis. Furthermore, the algorithm described above does not identify the *true* over-constraining set. It omits constraints that contribute to the over-constraint but whose removal would require loop or chain analysis to solve the constraint system. We are currently working to extend the algorithms to handle these cases.

5.2 Plan stepper and debugger

Since a constraint solution plan is a procedural program that is generated automatically, we are exploring the creation of tools, similar to conventional programming tools, for assisting users in understanding what the program does and how it was generated. One such tool is a *plan debugger* which will provide facilities found in debuggers for procedural programming languages: stepping forward and backward, setting breakpoints, and examining the status of constraints and constrained entities.

A plan debugger uses the already-generated plan to allow a user to replay the plan in single-step mode and review the choices made by the constraint solver when it was generating the plan. The constraint solver may be able to solve more than one constraint at a time, but "arbitrarily" chooses one of the possible constraints at each choice point. These choices reflect the fact that a plan is one possible traversal of the dependency graph. A user may also explore alternative traversals of the dependency graph by modifying the choices made by the constraint solver. Exploring alternative solution orders may give the user insight into the nature of conflicting or unsatisfiable constraints. The debugger calls on the solver to add a new branch to the plan if the user's new choice has not been explored before.

The debugger may also contain an explanation facility that reviews the solution steps and the choices made by the system to explain to the user an interim solution state and how that state was reached.

5.3 Parameterizing under-constrained models

An under-constrained model may be parameterized by identifying a set of independent model parameters that can be used to determine, completely and irredundantly, the behavior of the model. These parameters represent the degrees of freedom remaining in the model after all of its constraints have been satisfied.

In the graphics and geometric modeling domains, GCE adds "defaulting" constraints to a model whenever it runs out of constraints to solve and there are still some remaining degrees of freedom in the model [Kramer, 1992b]. Each defaulting constraint consumes one degree of freedom of a geometric entity. Defaulting constraints allow GCE to generate a solution to the now fully constrained model and tell the user the number of remaining degrees of freedom that may be consumed by additional constraints. These constraints are captured in the solution plan.

5.4 Improving computational efficiency

The dependency graph of a constraint model specifies a partial order for satisfying the constraints in the model. This partial order can be used to parallelize the solution of the constraint model. Subgraphs in a dependency graph, which correspond to subplans in the solution, can be traversed in parallel and provide coarse grain parallelism.[5] As in dataflow models of parallelism, fine grain parallelism can be identified because the solution of a constraint may proceed as soon as its predecessors in the dependency graph have been satisfied.

The dependency graph may also be used to minimize regeneration of a solution plan when a constraint is added or removed from a model. Graph traversal algorithms can identify portions of the dependency graph that are not affected by the addition or removal of a constraint from a model. The new solution plan reuses the unaffected portion of the previous solution plan (*i.e.*, the unaffected constraints are satisfied) and proceeds to solve the remaining constraints.

[5] Subgraphs result from the hierarchical strategies of chain analysis and loop analysis used in GCE. These are not discussed in this paper due to space limitations.

6 Conclusion

Constraint dependency information can be exploited in a number of areas: debugging and explanation of constraints, consistency analysis, parameterization of models, and computational optimization. As such, the explicit representation of this information in a dependency graph is valuable. The results obtained so far look promising and warrant further study.

References

[AIJ, 1992] *Artificial Intelligence* (Special Volume: Constraint-Based Reasoning), volume 58, December 1992.

[Belleville, 1992] Laureen Belleville. Applicon's intelligent approach to modeling. *Computer Graphics World*, page 17, November 1992.

[Borning, 1979] Alan H. Borning. *ThingLab: A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford University, July 1979.

[Breuer and Friedman, 1976] Melvin A. Breuer and Arthur D. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, Rockville, MD, 1976.

[Kramer, 1990] Glenn Kramer. Solving geometric constraint systems. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 708–714, Boston, MA, August 1990. American Association for Artificial Intelligence, MIT Press.

[Kramer, 1992a] Glenn Kramer. *Solving Geometric Constraint Systems: A Case Study in Kinematics.* MIT Press, Cambridge, MA, 1992.

[Kramer, 1992b] Glenn A. Kramer. A geometric constraint engine. *Artificial Intelligence*, 58:327–360, December 1992.

[Mills, 1992] Robert Mills. Terms of endearment. *Computer-Aided Engineering*, pages 48, 50, 54, October 1992.

[Pabon *et al.*, 1992] Jahir Pabon, Robert Young, and Walid Keirouz. Integrating parametric geometry, features, and variational modeling for conceptual design. *Systems Automation: Research and Applications*, 2:17–36, 1992.

[Sutherland, 1963] Ivan E. Sutherland. *Sketchpad: A Man-Machine Graphical Communication System.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1963.

[Tee, 1992] Cynthia Tee. Human factors in geometric feature visualization. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1992.

Implementing Computational Systems with Constraints

Claude Kirchner **Hélène Kirchner** **Marian Vittek**

INRIA-Lorraine & CRIN
615, rue du Jardin Botanique, BP 101
54602 Villers-les-Nancy Cedex FRANCE
E-mail: kirchner@loria.fr, vittek@loria.fr

Abstract

The paper presents a framework to describe, experiment and study the combination of different computational systems including the constraint solving paradigm. Computational systems are interpreted in a first-order setting thanks to an evaluator that rewrites formulas.

1 Introduction

Logic programming in a broad sense, theorem proving and constraint solving have in common their foundation on a computational system, described by the syntax of formulas, a set of axioms, a set of deduction rules whose application is governed by some strategies, altogether with one or a class of interpretations. All these activities may be formulated as instances of a common schema that consists of applying deduction rules on formulas with some strategy, until getting specific forms. In logic programming the emphasis is on efficiency in reaching a successful solved form characterising an answer, in theorem proving on completeness in detecting trivially false or true formulas, and in constraint solving on getting some predefined solved form of a constraint.

We claim that this similarity is a major advantage for the design of safe programming environments where programs in high-level programming languages can be developed and some of their properties can be proved. Moreover we believe that the implementation of such an environment can rely on a very simple theoretical setting: first-order equational logic. This means that we see formulas, sets of formulas and proofs as first-order terms. We see a deduction rule as a rewrite rule or a transition rule, that is a rewrite rule applying at the outermost position of such terms. Application of deduction rules is rewriting, possibly under some conditions. We think that this framework has several advantages:
— it is conceptually simple.
— we benefit from the work on rewriting and efficient implementation techniques.
— we can define rewriting in the language itself, so bootstrap the system.
— more important, we can reason about computational systems, for instance prove abstract properties of the deduction rules like termination or confluence with rewrite-based techniques. We can also reason about combination and enrichment of computational systems and build new paradigms in a modular way, once we know how systems theoretically interact.

To validate these ideas, we develop a prototype[1] called ELAN, to describe, experiment and study the combination of different computational systems that provide basis for logic programming paradigms. Our favorite starting example, developed in this paper, is the combination of constraint solving with computational systems based on rewriting and narrowing.

[1] This prototype has been partly designed in collaboration with the DEMONS group in Orsay and supported by GRECO PAOIA of CNRS, and Esprit BRA CCL.

2 Conceptual setting

The conceptual setting is given by the notion of computational system whose evaluation mechanism is based on application of transition and rewrite rules. This provides uniformity of the execution principle which is first-order. The notations and definitions used in this paper are consistent with [DJ90, JK91].

In our approach, a **computational system** is given by a syntactic part and a computational part.

- The syntax is defined by
  - a class *Sign* of signatures; each signature $\Sigma \in Sign$ defines a set of sort symbols, a set of function symbols and a set of predicate symbols.
  For each signature $\Sigma \in Sign$,
  — $stat(\Sigma)$ denotes the sets of statements that are sentences (well-formed formulas) built on this signature.
  Each pair $P = (\Sigma, \Gamma)$ where $\Sigma \in Sign$ and $\Gamma \in stat(\Sigma)$ defines an axiomatic theory.
  — $quer(\Sigma)$ denotes the queries that are sentences built on the signature Σ.

- The computational part is defined by deduction rules and their strategy of application. These deduction rules are aimed to transform queries into "simpler" queries in the sense that they can easily be solved in the axiomatic theory $P = (\Sigma, \Gamma)$. Indeed it is assumed that the deduction rules and their application strategy have been proved correct and complete w.r.t. an adequate semantics. We do not focuss here on this theoretical point but only consider examples where such correctness and completeness results are available.

  A **computation** is a sequence of terms $t_0 \Rightarrow_{r_0} t_1 \cdots \Rightarrow_{r_{n-1}} t_n$, where each t_{i+1} results from the application of the transition rule r_i on t_i. A sequence of names of transition rules is a strategy. More precisely, **a strategy** is a set of such sequences described by a regular expression built on the names of deduction rules using concatenation, iteration and selecting operators. This regular expression (strategy) describes the set of all admissible computations. A computation $t_0 \Rightarrow_{r_0} t_1 \cdots \Rightarrow_{r_{n-1}} t_n$ is admissible w.r.t. the strategy s iff the word "$r_0 \cdots r_{n-1}$" is in the language $SLan(s)$ described by the regular expression s. t_n is called the result of this computation w.r.t. s. From this definition we see that more than one computation starting from the initial term t_0 can belong to $SLan(s)$; in this case the strategy of computation s is nondeterministic and its result is the multiset of results for all admissible computations. On the other hand, when there is no admissible computation starting from t_0, the result of the strategy s is an empty multiset.

For example, a **Horn clause computational system** is defined by chosing *Sign* as the class of first-order signatures, *stat* as sets of formulas of the form $(\forall X : A \Leftarrow B_1, \cdots, B_n)$, *quer* as formulas of the form $(\exists X : G_1, \cdots, G_n)$ where X is a set of variables, A, B_i, G_k are atoms. SLD resolution can be described using deduction rules that produce, from the negation of the query, the empty clause which is trivially unsatisfiable. A correct and complete strategy must include all computations leading to the empty clause.

Another example is an **equational computational system** defined by choosing *Sign* as the class of first-order signatures with "$=$" and "$\rightarrow$" being the only predicate symbols, *stat* as all sets R of universally quantified formulas of the form $(\forall X : l \rightarrow r)$, such that R is a convergent term rewriting system, *quer* as formulas of the form $(\exists X : g = d \parallel S)$, in which S is a possibly empty conjunction of equations denoted $\bigwedge(t =_\emptyset t')$, which are solved with syntactic unification (also called unification in the empty theory). The basic narrowing process [JK91] can be used to solve such queries, provided that R is a convergent term rewriting system. Let us briefly explain this process. Starting from a query of the form $(\exists \emptyset : g = d \parallel \top)$, where $\top$ denotes the empty conjunction of equations, the basic narrowing process tries to unify a subterm of g at position ω with a left-hand side of a rewrite rule $(l \rightarrow r) \in R$, taking care that its variables are disjoint from those of g. This gives rise to a unification problem $(g_{|\omega} =_\emptyset l)$ and a new equation $(g[r]_\omega = d)$. So the query $(\exists X : g = d \parallel S)$ is transformed into another one $(\exists X \cup Var(l \rightarrow r),\ g[r]_\omega = d \parallel S \wedge (g_{|\omega} =_\emptyset l))$ such that the two systems of equations $(\exists X : g = d \wedge S)$ and $(\exists X \cup Var(l \rightarrow r),\ g[r]_\omega = d \wedge S \wedge (g_{|\omega} =_\emptyset l))$ have the same set of solutions in the theory defined by R. Indeed a query $(\exists X : g = d \parallel S)$ has an obvious solution when g and d are syntactically unifiable and this is exactly the solution of the system $S \wedge (g =_\emptyset d)$. Note that a successful computation transforms a problem in the theory R into a unification problem in the empty theory. Indeed there is a lot of undeterminism in the choices of the rewrite rule $(l \rightarrow r)$ and the

position ω. All possible choices have to be explored when completeness of the returned set of solutions is required (see [NRS89]).

Accordingly, basic narrowing is described with two deduction rules:

Narrow
$(\exists X, g = d \parallel S)$
$\rightarrowtail$
$(\exists X \cup Var(l \rightarrow r),\ g[r]_\omega = d \parallel S \wedge (g_{|\omega} =_\emptyset l))$
if $l \rightarrow r \in Variants(R)$ and $S \wedge (g_{|\omega} =_\emptyset l)$ satisfiable

Block
$(\exists X, g = d \parallel S)$
$\rightarrowtail$
$(\exists X, \top \parallel S \wedge (g =_\emptyset d))$
if $(S \wedge g =_\emptyset d)$ satisfiable

In this example, S is a conjunction of equational constraints solved in the empty theory. In general, S could be solved in any equational theory, for instance with associativity and comutativity axioms, or even in a specific algebra, for instance a boolean algebra. The previous deduction rules are actually parameterized by the constraint solving process. The next step is to describe it also as a computational system.

3 Constraint solvers as specific computational systems

The description of constraint solving using rule-based algorithms as in [Com91, JK91], allows easier correctness and completeness proofs of constraint solvers, partly thanks to the explicit distinction made in this approach between deduction rules and control. This is also the starting point for incorporating constraint solving in our framework. A constraint solver for symbolic constraints such as equations, inequations or disequations on terms is also viewed as a computational system aimed at computing solved forms for the class of considered formulas called constraints.

For instance, **unification** in commutative theories is a constraint solving process that can be described as the computational system where *Sign* is the set of first-order signatures with equality in equivalence classes, denoted $=_C$, as the only predicate (C stands for commutative), *stat* is empty, *quer* is the set of formulas of the form $(t_1 =_C t_2)$. Deduction rules have been proposed and proved correct and complete w.r.t. the computation of a complete set of C-unifiers (see [JK91] for instance). These rules transform conjunctions of equations of the form $(P \wedge s =_C t)$ and include for instance the following ones for a commutative operator $+$:

ComDecompose
$P \wedge s_1 + s_2 =_C t_1 + t_2$
$\rightarrowtail$
$P \wedge s_1 =_C t_1 \wedge s_2 =_C t_2$

ComMutate
$P \wedge s_1 + s_2 =_C t_1 + t_2$
$\rightarrowtail$
$P \wedge s_1 =_C t_2 \wedge s_2 =_C t_1$

In general, a language of constraints is defined by a class of signatures defining the syntax of the constraints, the set of queries which are the constraints to solve in this language and a constraint solver. We are interested here in constraint solving processes that can be described with transition rules that compute solved forms of constraints. This includes for now constraint languages built from elementary constraints that may be equations (as above), disequations [Com91], inequations [Com90] on terms expressed with simplification orderings, membership constraints [CD91].

This view has several advantages over constraint solving systems where solvers are encapsulated in black boxes.

1. Although completeness may be in some cases, like syntactic unification, proved for any strategy, the solved forms are in general reached more efficiently with smart choices of rules. Through the design of rules and strategies, the developer has a direct access to the constraint solver and this point is crucial for discovering and experimenting new solvers.
2. To be able to formalize constraint solving by rewrite rules also makes termination proofs easier and eventually partly automated. This also opens the door to non-trivial applications of termination proof techniques.
3. Another interest of this point of view is the conceptually easy combination of constraint solving with other computational systems. Indeed a computational system for a language involving now formulas with constraints ($\varphi \parallel c$), where φ is a formula (a Horn clause for instance) and c is a constraint, is incrementally built by importing a computational system for the constraint language in which c is expressed. Statements are sets of formulas with constraints, like in $CLP(X)$ [JL87]. It is yet possible to define transition rules on formulas with constraints, but we shall see later on in Section 5, that such a combination may also generate delicate problems.

4 The ELAN interpreter

ELAN is an environment for designing computational systems, and evaluating queries written in this system. An executable entity for the ELAN interpreter is given by:

A computational system description that provides the syntax, the deduction rules and the strategy of the computational system which is defined.

Specifications that are axiomatic theories written in the syntax of the defined computational system and defining their own signature and set of axioms.

Queries that express the problem to solve in a given specification and can be seen as input values for the computation.

A specification interpreted in a given computational system produces an executable program, which is able to evaluate a query w.r.t. the given computational system.

To achieve our goal of a uniform yet flexible system, the most delicate points were:

- to express the syntax of any computational system that the user wants to define. 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 corresponds to a computational system. It defines its own signature, that is the symbols used to express the syntax of statements and queries, but also function symbols used to express transition rules. It defines its own transition rules and rewrite rules useful to express functions used by transition rules. It defines also strategies for applying these transition rules, that may be deterministic or non-deterministic.
- to express the evaluation mechanism of programs for given inputs, i.e. the mechanism for solving the queries. To evaluate a query, the ELAN interpreter repeatedly applies the transition rules according to the strategy, until the strategy is exhausted. A transition rule is implemented as a rewrite rule that can be applied only at the outermost position of a term and whose application is conditionned by a kind of local assignment. Since our choice is to describe computations by transition rules controlled by strategies, a convenient and powerful language to express strategies is needed. ELAN provides in particular don't-know and don't-care non-determinism, so the possibility to express backward and forward chaining for the deductions.

In the following part of this section, we illustrate the capabilities of ELAN on an example. Let us consider again the case of an equational computational system with narrowing as deduction process. Programs in this simple language are algebraic specifications consisting in the definition of a signature and axioms given as rewrite rules. In Figure 1, a specification of lists is given.

```
specification rwspec
 signature
   Vars: X, Y, Z
   Ops:  nil, cons:2, append:2, a, b, c
 rwrules
   append(cons(Y,Z),X) -> cons(Y,append(Z,X)),
   append(nil,X) -> X
end of specification
```

Figure 1:

A query is then a single equation between two terms built on this signature.

A computational system description has to define the syntax of the specification and of the query as well as how the query is solved. In a computational system description, all the notions of terms, equations, systems of equations have to be specified explicitly. Thanks to modularity, it is indeed possible to reuse a library of modules with mostly used constructions.

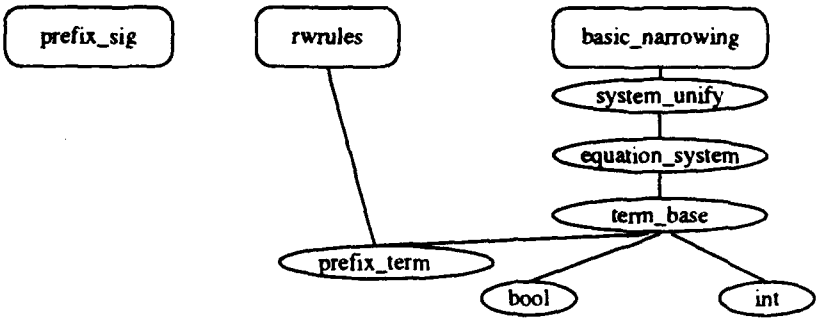


Figure 2:

For instance, defining the signature part of a specification needs to define what are functional symbols, variables, arities, as well as the syntactic form of a function declaration in the specification. This is done in a module *prefix_sig*. Similarly, defining the rewrite rules part of a specification needs to define a notion of term (here in the module *prefix_term*) and the notion of rewrite rule (in the module *rwrules*). The module *prefix_term* is reused by the module *basic_narrowing* where a query and its execution are defined. Figure 2 shows all modules used in the narrowing example with their import relations.

We do not explain here the construction of the concrete syntax of a specification and how it is processed by the system (this is described in [Vit93]), but rather focus on the processing of queries.

4.1 Transition Rules

As already said, the main ELAN mechanism to process a query is application of transition rules. The syntax for transition rules in ELAN is:

```
<trans rule> ::= transition rule <name of rule> for <type of terms>
                 declare <local variable declarations>
                 body
                   <rule body>
                 end of rule
```

where

```
<rule body> ::=  <term> => <term>
              |  <rule body> where <variable> := ( <strategy> ) <term>
```

The **where** construction is a local assignment which assigns to the variable **<variable>** the result of applying the strategy **<strategy>** to the term **<term>**. If the strategy is nondeterministic, then all results are considered. More precisely, let r be a rule body, x a local variable, s a strategy and t a term. Then the rule

$$r \text{ where } x := (s)\ t$$

conceptually represents the set of rules, all with the same name, that are instances of r of the form $(x \mapsto t')r$ for any term t' resulting from the computation of t with respect to s.

A first example of transition rules, without the **where** construction, is given by the implementation of the rules **ComDecompose** and **ComMutate**, for which the translation in ELAN is straightforward:

```
transition rule ComDecompose for system
declare P : system;
        s_1,s_2,t_1,t_2 : term;
body
  P & s_1+s_2 = t_1+t_2        =>        P & s_1 = t_1 & s_2 = t_2
end of rule

transition rule ComMutate for system
declare P : system;
        s_1,s_2,t_1,t_2 : term;
body
  P & s_1+s_2 = t_1+t_2        =>        P & s_1 = t_2 & s_2 = t_1
end of rule
```

In general, functions in ELAN can be written in *mixfix notation*; *in the last rule we have used the* operators "& + =" in infix form.

The rule **Narrow** from the narrowing process provides a more complex example, that illustrates several features of ELAN. It will be written as:

```
IN rwrules: FOR EACH L,R:term SUCH THAT rwrule(L,R) : {
  transition rule Narrow for b_narr_state
  declare cc,c1   : system;
          g,d,l,r : term;
          sigma   : substitution;
          oc      : occurrence;
  body
    (g = d || cc)        => ( g[r] at oc = d || c1)
      where c1 := (unifys) g at oc = l & cc
      where oc := (occ) non-var-occ(g)
      where l :=  () sigma(L)
      where r :=  () sigma(R)
      where sigma := () rename-subst(L, g = d & cc)
  end of rule
}
```

The mixfix operators used in this rule have the following meanings: applying a substitution *sigma* on a term t is written "*sigma*(t)"; the subterm of t at position *oc* is written "t *at oc*"; the term produced by replacing a subterm of t at position *oc* by the term r is written "$t[r]$ *at oc*". All these functions are imported from other modules.

The quantification **IN rwrules: FOR EACH L,R:term SUCH THAT rwrule(L,R): { <rule> }** means that **<rule>** will be generated for all instances L,R such that **L->R** is a rewrite rule defined in the specification.

All these rules have the name **Narrow**. For instance two such rules will be generated in the list specification given in Figure 1.

During the application of this transition rule, the variables are successively assigned to the following values: *sigma* to a renaming substitution which introduces fresh variables not occurring in the current system; *l* and *r* to left and right-hand side of the rewrite rule `L->R` after variable renaming; *oc* is any non-variable position in the term g (occ is a implemented by a non-deterministic strategy); *cl* is a solved form of the unification problem $g_{|oc} = l \;\&\; cc$. The strategy **unifys** returns a solved form just in the case where the original system is satisfiable. It may be worth emphasizing that the transition rule only applies when every local variable has got a result different from the empty set. Else application of the transition rule fails.

For the sake of completeness, we can write in the same way the rule **Block**.

```
transition rule Block for b_narr_state
declare SF,cc: system;
        g,d: term;
body
  (g=d || cc)   => ( top || SF )
        where SF := (unifys) g=d & cc
end of rule
```

4.2 Strategies

Coming back to the narrowing example, we can observe that after n applications ($n \geq 0$) of the rule **Narrow** followed by one application of rule **Block**, the resulting term is of the form $(\top \parallel S)$ where S provides a solution for the starting equational query. On the other hand, the completeness result for narrowing states that a complete set of R-unifiers is obtained by gathering all solutions that can be computed as a result of n applications of rule **Narrow** for some $n \geq 0$, followed by an application of **Block**. Accordingly, the set of successful computations, that is computations leading to a solution, is described by the following regular expression:

$$\textbf{(Narrow)}^* \cdot \textbf{Block}$$

where the $\cdot$ is the concatenation operator, and $()^*$ is an iteration operator. So we consider this expression as describing a correct and complete strategy for narrowing.

A partial concrete syntax for strategies in ELAN is:

```
<strategy> ::= iterate <strategy> enditerate
           |   dont know choose( <list of <strategy>>  )
           |   dont know choose( <list of <rule name>>  )
           |   <strategy> <strategy>
```

In this syntax, concatenation is omitted, the constructor **iterate** corresponds to the iterator $()^*$ and the **dont know choose** constructor corresponds to a selecting operator. Note that names of rules are always encapsulated in the selecting operator. This is because the rule name represents in general a set of rules, so a selecting operator has always to be specified. The full ELAN strategy language is of course richer and includes constructions for dont-care strategies, which is interesting especially from the efficiency point of view and to formalize forward chaining computations.

Since narrowing can be described by a regular expression **(Narrow)*** $\cdot$ **Block**, the strategy for our example can be implemented as follows in ELAN:

```
strategy b_narrow
  iterate
    dont know choose(Narrow)
  enditerate
  dont know choose(Block)
end of strategy
```

This strategy will generate application of the transition rule **Block**, then of **Narrow** followed by **Block**, then of twice **Narrow** followed by **Block** and so on. This leads to a correct implementation of narrowing in ELAN.

Regular expressions can seem to be very weak to express computational strategies, but their power is largely increased by using the **where** construction inside the rules and by the definition of admissible computations given in Section 2. According to this definition, the **dont know choose** constructor has to try all possible applications of a transition rule in the list given as argument. In our example, this is the way to exhaustively attempt to apply narrowing with all rewrite rules in R and all positions in the query term. The **iterate** constructor then successively tries all possible numbers of iterations. In the ELAN interpreter, backtracking is used to implement a computation following this kind of strategy.

Beyond the case of narrowing, we succeeded for instance to specify commutative unification, rewriting, and a simple PROLOG interpreter.

4.3 Flexibility

ELAN supports computational system descriptions with high level of modularity. Modules are non-parametric and their visibility graph is a directed acyclic graph. Each module can use signatures from modules explicitely introduced in an import part. Aliasing of operators is permitted, so an operator can have more than one name; for example it can be used in mixfix and prefix notation at the same time. Direct importation with aliasing allows providing all operations usual in such modular systems, like renaming, masking and so on.

The flexibility of ELAN can be briefly illustrated on two examples.

— We can straightforwardly replace the syntactic unification in narrowing procedure by unification in another theory. We just need to replace the module *system_unify* by a module providing unification in the desired theory. This can be compared to the $CLP(X)$ framework that could be implemented in ELAN. Each instance would be provided by the choice of a computational system for constraint solving in the computation domain.

— We can also easily change the syntactic form of specification. For example, replacing the module *prefix_sig* by a module *mixfix_sig*, adding a module *mixfix_term* and some minor modifications of *basic_narrowing*, provides mixfix notation at the specification level.

5 Combining computational systems

Up to now we have expressed in a uniform framework constraint solving and deduction and achieved a high level of modularity. Our objective of designing and experimenting with new computational systems leads us to consider different combination problems. Let us mention two kinds of theoretical problems that arise in this context:

- Putting together a computational system for equality and constraint solving like unification leads to interesting but non-trivial problems from the point of view of their properties [KKR90]. For instance from two oriented equality with constraints, a new equality can be deduced using the deduction rule

$$\begin{array}{l}\textbf{Deduce}\\ (g \to d \parallel c'), (l \to r \parallel c)\\ \longmapsto\\ (g[r]_m \simeq d \parallel c \wedge c' \wedge (g_{|m} =_\emptyset l))\\ \text{if } c \wedge c' \wedge (g_{|m} =_\emptyset l) \text{ satisfiable}\end{array}$$

  Unfortunately this deduction rule does not capture all possible equational deductions. Consider for instance the set of function symbols $\mathcal{F} = \{a, c, f, g\}$, and two rewrite rules with equational constraints:

$$\begin{array}{l} g(a) \to c\\ f(x) \to a \quad \parallel \quad x =_\emptyset g(y)\end{array}$$

  Deduce does not apply although

$$a \leftarrow f(g(a)) \to f(c)$$

  is a correct deduction performed with the given rewrite rules. This problem is solved in a general way in [KKR90] by restructuring rules and constraints. Such restructuration has been shown useless with additional syntactic restrictions on formulas. For instance it has been proved that Deduce is complete with a limited class of initial constraints [BGLS92, NR92].

- Putting together two constraint solvers in two different first-order structures like for instance booleans and a quotient term algebra cannot be reduced to a blind use of both constraint solvers. Typically interactions between theories may appear and must be solved [BS92, KR92].

Such problems are yet an active field of research.

6 Further perspectives and conclusion

As already emphasized, we expect the system to provide a uniform framework for describing computational systems, experimenting new combinations and performing proofs of their properties. Completion-based approaches implemented in theorem provers like RRL or OTTER, to prove properties such as confluence or consistency of enrichments, can also be designed in the conceptual setting of transition rules and strategies [Bac91]. Concepts like critical pairs criteria, that appear very useful in the design of efficient provers can be formalised with constraints [KKR90]. This view is not to obtain efficient theorem provers at once, but rather to allow for experiments and designs of better proof strategies.

A complementary direction is worth considering to get high efficiency: this is the parallelisation of deduction obtained for free through parallelisation of rewriting (see [KV92, Mes92]). This argument that needs further investigations is an additional point in favor of the choice of the evaluation mechanism of ELAN.

Indeed some aspects of this work can be compared with other logical frameworks like ELF, proof development systems like NuPRL and interactive theorem provers like Isabelle, Clam or 2OBJ. However we think that, because of its emphasized ability to express, execute and combine computational systems and the crucial role of constraint solving, ELAN provides interesting and original features.

References

[Bac91] L. Bachmair. *Canonical equational proofs.* Computer Science Logic. Progress in Theoretical Computer Science. Birkhäuser Verlag AG, 1991.

[BGLS92] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In *Proceedings 11th International Conference on Automated Deduction, Saratoga Springs (NY, USA)*. 1992.

[BS92] F. Baader and K. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In *Proceedings 11th International Conference on Automated Deduction. Saratoga Springs (NY, USA)*, 1992.

[Com90] H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4), 1990.

[Com91] H. Comon. Disunification: a survey. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 9. MIT Press, Cambridge (MA, USA), 1991.

[CD91] H. Comon and C. Delor. Equational formulas with membership constraints. Technical report, Laboratoire de Recherche en informatique, Mar. 1991. To appear in Information and Computation.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers North-Holland, 1990.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th POPL conference*, Munich (Germany), 1987.

[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. 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.

[KR92] H. Kirchner and C. Ringeissen. A constraint solver in finite algebras and its combination with unification algorithms. In *Proc. Joint International Conference and Symposium on Logic Programming*, 1992.

[KV92] C. Kirchner and P. Viry. Implementing parallel rewriting. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, volume 590 of *Lecture Notes in Artificial Intelligence*, pages 123–138. Springer-Verlag, 1992.

[Mes92] J. Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Proceedings 3rd International Conference on Algebraic and Logic Programming, Volterra (Italy)*, volume 632 of *Lecture Notes in Computer Science*, pages 158–200. Springer-Verlag, September 1992.

[NR92] R. Nieuwenhuis and A. Rubio. Basic superposition is complete. In B. Krieg-Brückner, editor, *Proceedings of ESOP'92*, volume 582 of *Lecture Notes in Computer Science*, pages 371–389. Springer-Verlag, 1992.

[NRS89] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7(3 & 4):295–318, 1989. Special issue on unification. Part one.

[Vit93] M. Vittek. ELAN User's manual. In preparation.

Aggregation in Constraint Databases

(Preliminary Report)

Gabriel M. Kuper
ECRC
Arabellastr. 17
D-8000 München 81
Germany.

Abstract

We discuss the issues that arise when we add aggregation to a constraint database query language. One example of the use of aggregation in such a context is to compute the area of a region in a geographic database. We show how aggregation could be added to the query language, tuple calculus, and discuss the problems that arise from the interaction of aggregate operators and constraints.

1 Introduction

[KKR90] proposes the use of constraint query languages as a natural way of combining relational databases with constraint formalisms (see also [Kuper90] [Revesz90]). One important aspect of relational databases that is not discussed in that paper is the use of aggregation. Typical aggregate operators in relational databases are: Sum, Average, Count. In a spatial database, the simplest analogue is computing the area of an object. More complicated aggregate operations, such as averaging an attribute over a given region, could be useful.

In this paper, we outline a method to add aggregation to the languages of [KKR90]. Our approach resembles the approach taken by Klug [Klug82] in the framework of traditional relational database systems. The current

paper has two main parts: We first illustrate (mostly by examples) how Klug's approach can be generalized to the constraint framework. We then discuss the problems that arise from the interaction between aggregation and constraints. These problems concern mainly closure under aggregate operations. We conclude with a description of possible directions in which research could be pursued to deal with these problems.

2 Aggregation with Constraints: An Example

The main idea behind Klug's approach is to enable the user to group the tuples being aggregated in all possible ways, without introducing the need to have relations with duplicates.

We will not give here a formal description of Klug's query languages. However, we hope that the key ideas will be clear from the following example, even for the reader who is not familiar with Klug's paper.

Example 1 Let $R(n, x, y)$ be a ternary relation containing a database of rectangles. The semantics of this relation are as follows: Tuple (n, x, y) is in R iff (x, y) is a point in the rectangle with name n. This is the same example as in [KKR93]. As explained there, we represent each rectangle in the database by constraints, rather than listing all the points explicitly.

There are various ways in which we could apply the *area* function to this relation:

1. Suppose we want to compute the area of each rectangle separately. In other words, we want to compute a relation $S(n, a)$, where a is the area of the rectangle with name n. This query is written as follows:

$$R\langle 1, \text{area}_{2,3}\rangle$$

   The meaning of this notation is as follows. We apply, to the relation R, the aggregation operator that groups the tuples on their first argument, and then applies the aggregate operator $\text{area}_{2,3}$ to the second and third column.[1]

[1] For technical reasons we have a version of the aggregate operator for each pair of

2. Now suppose that we are interested in the total area covered by the points of the database, counting points that are in several rectangles only once. We can do this, by projecting R onto the second and third columns, thus generating a flat representation of the data without the rectangle names. We then apply aggregation to the result. Formally, the query is:

$$(\Pi_{2,3}(R))\,\langle \emptyset, \mathrm{area}_{1,2} \rangle$$

Note the first argument of the aggregation is $\emptyset$. This means that the result has only one column, and this column contains the result of the aggregation.

3. Finally, suppose that we want to compute the total area, but this time counting points that are in several rectangles multiple times. To do this, we first compute the relation S as in the first example, and then sum the second column of the result. We can do this using the *sum* aggregate operator on traditional relational databases. We can apply such an operator to a constraint database whenever the relevant column contains only a finite number of elements. Formally, the query is:

$$(R\,\langle 1, \mathrm{area}_{2,3} \rangle)\,\langle \emptyset, \mathrm{sum}_2 \rangle$$

An important aspect of this example that we would like to stress is that, as in [KKR93], while the example we gave is of a database of rectangles, we do not actually made use of this fact. In other words, the queries in this example will work, without any changes, for any database that consists of geometric objects that have a well defined area.

3 Constraint Query Languages with Aggregation

In this section we briefly describe how to generalize [Klug82] to constraint database. Our basic setting is similar to Klug's. We have a set

$$\mathrm{Agg} = \{f_\alpha, f_\beta, \ldots\}$$

distinct columns. The version of the *area* function is generated automatically by the tuple calculus to algebra conversion algorithm. This means that the multiple versions of each aggregate operator are in fact hidden from the user.

of aggregate functions. Each function is from the set of all relations to the domain D of the constraints. For example, if we are dealing with polynomial constraints, the range of the aggregate functions must be the reals. We have a "uniformness" condition, as in Klug, that basically says that if we have an aggregate function, say $\text{area}_{1,2}$ in *Agg*, then we have "similar" functions $\text{area}_{i,j}$, for all $i \neq j$.

The first point in which our approach differs from Klug's, is that aggregate functions can have more than one subscript. In the relational model the aggregate functions are unary: average, sum etc. In constraint databases we can have aggregate operators with higher arities: For example binary (e.g., area), ternary (e.g., volume), etc.

Another point where care is needed is the fact that most of the languages described in [KKR90] use constraints over an unbounded domain (integers, reals, etc.). This means that we could have objects in the database whose are is infinite. We can avoid this problem by assuming that the database instances are bounded. We expect that in real systems this will usually be the case. (Note that a database instance could still represent an infinite set of points, but in a bounded region of space). This is a point where more research is needed, since this problem does have some resemblance to issue of safety in relational databases, and our solution is not completely satisfactory.

For our purposes it will, however, suffice, and we can now define a tuple calculus and algebra for constraint databases with aggregation. We omit the formal details of the construction of the query languages and the proof of their equivalence, since these are fairly straightforward extensions of Klug's work. We hope that the example in the previous section gives a reasonable idea of how the relational algebra is defined.

The most critical difference between constraint and relational databases is a consequence of the requirement that the language be closed. The rest of this paper will discuss the consequences of this requirement.

4 Closure under Aggregate Operations

So far, we have discussed the relational algebra and calculus on constraint databases in terms of the underlying semantics. The problem is that the result of a query must be representable by a finite set of constraints, and there has to be an effective way to compute this representation. It turns out

that for many interesting classes of constraints this is not the case.

Example 2 Consider the language of linear inequality constraints. Let $R(x, y, z)$ be a ternary relation, with an instance that consists of exactly one generalized tuple:

$$x - y - z \leq 0 \wedge x \geq 0 \wedge y \geq 0$$

Suppose that, for each z, we want to compute the area of the region

$$\{(x, y) \mid R(x, y, z)\}$$

Formally, we want to evaluate the query

$$R\langle 3, \text{area}_{1,2}\rangle$$

The result is a binary relation $S(z, t)$. Unfortunately, the result S contains all tuples (z, t) for which $z^2 = 2t$. Clearly, in order to represent this instance we need quadratic constraints, i.e., the relational algebra with linear inequality constraints is not closed under aggregation by *area*.

If we think about this example, this result is actually quite reasonable. Computing the area of a region is essentially integration. Since the integral of a linear function is a quadratic function, it is quite reasonable to expect that quadratic functions are needed to express the result. More surprisingly closure under aggregation by area does not hold even for dense order constraints.

Example 3 Let $R(x, y, z)$ be a ternary relation, with an instance consisting of the generalized tuple

$$0 < x < y < z < 1$$

Once again, we have the query

$$R\langle 3, \text{area}_{1,2}\rangle$$

whose result is binary relation $S(z, t)$ containing all tuple (z, t) for which $z^2 = 2t$. Once again, quadratic constraints are needed to express the result.

What about polynomial constraints? It turns out that even these are not closed under aggregation by *area*.

Example 4 Let $R(x, y, z)$ be a ternary relation, with the instance consisting of

$$\begin{cases} x^2 + y^2 \leq 1 \\ x \leq z \\ x, y, z \geq 0 \end{cases}$$

Once again, what is the result of the query

$$R\langle 3, \text{area}_{1,2} \rangle \quad ?$$

We get a relation $S(z, t)$, containing all tuples (z, t) such that t is the area of the part of the unit circle above the x-axis, and between $y = 0$ and $y = 1 - z$. In other words, (z, t) is in the result iff

$$t = \int_0^{1-z} \sqrt{1 - x^2} dx$$

i.e., iff

$$t = \frac{1}{2}\left((1 - z)\sqrt{z(2 - z)} + \sin^{-1}(1 - z)\right)$$

Clearly, this cannot be expressed using polynomial constraints

5 Discussion

We have described here the consequences of adding aggregation operators to a constraint-based database query language. The main problem that arises is finding a finite representation of the result. Most interesting classes of constraints are not closed under aggregation by *area*. For example, to represent the area covered by linear constraints requires polynomial constraints, and the area covered by polynomial constraints needs transcendental constraints. We outline three approaches that could be pursued to handle this problem, namely, restricting the aggregation operators allowed, restricting the query language, and using a typed language in order to make the consequences of the closure problem less serious.

1. *Restricting the aggregation operators.* It is possible that some aggregation operators, for example *minimum* and *maximum* would have finitely

representable results, using the same class of constraints. One interesting research area is to characterize those aggregation functions with such a property, for the various classes of constraints.

This is of course not a complete solution of the problem, since certain operators, such as *area* would not be in this class, and they are very important in some applications.

2. *Restricting the query language.* It is possible that for many applications, the full power of the relational query language with aggregation is not required. For example, in a GIS application, we might want to compute the area of several features in a map. As this is a finite set of values, this could be represented by a standard, finite, relation, avoiding the closure problem.

   There are, however, applications for which this solution does not suffice. For example, in a database that combines spatial and temporal components, we might want to determine how the area of some feature varies with time. Using a finite relation to store the result loses the advantages of the constraint-based approach. On the other hand, this is precisely the type of query that gives rise to the closure problem.

3. *Using a typed language.* In this approach we do not try to avoid the closure problem, but rather to mitigate its consequences.

   Consider the *area* function. While, superficially, the domain and range of this function are both the set of real numbers, if we take in to account the dimensions of the underlying physical units, the domain and range are in fact different.[2] We can therefore have different classes of constraints that are allowed, depending on the types of the variables. For example, we could have linear constraints on length variables, but polynomial constraints connecting length and area variables.

   In this way, we could isolate the places where the "harder" classes of constraints can appear, in such a way mitigating their effect on the efficiency of the query language.

[2] *In fact, when we said that the fact that the domain of the constraints is bounded implies that the area is also bounded, we ignored the fact that the bounds are different. Using a typed language would address this problem as well.*

Acknowledgments

I would like to thank Paris Kanellakis for useful discussions, in particular for providing some of the motivation for this paper.

References

[Dincbas88] M. Dincbas, et al. The Constraint Logic Programming Language CHIP. *Proc. Fifth Generation Computer Systems*, Tokyo Japan, 1988.

[JL87] J. Jaffar, J.L. Lassez. Constraint Logic Programming. *Proc. 14th ACM POPL*, 111–119, 1987.

[KKR90] P. Kanellakis, G. Kuper and P. Revesz. Constraint Query Languages. *Proc. 9th ACM PODS*, pp. 299–313, 1990.

[KKR93] P. Kanellakis, G. Kuper and P. Revesz. Constraint Query Languages. JCSS, to appear.

[Klug82] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *ACM Trans. on Database Systems*, 29 (3), pp. 699–717, 1982.

[Kuper90] G. Kuper. On the Expressive Power of the Relational Calculus with Arithmetic Constraints. *Proc. 3rd International Conference on Database Theory*, 1990.

[Revesz90] P.Z. Revesz. A Closed Form for Datalog Queries with Integer Order. *Proc. 3rd International Conference on Database Theory*, 1990.

[VanHen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

Constraint Satisfaction in Functional Programming

François Major
National Center for Biotechnology Information
National Library of Medicine
National Institutes of Health
Bethesda, MD 20894
major@ncbi.nlm.nih.gov

Marcel Turcotte and Guy Lapalme
Département d'Informatique et de Recherche Opérationnelle
Université de Montréal
Montréal, Québec, Canada
H3C 3J7
turcotm@medcn.umontreal.ca
lapalme@iro.umontreal.ca

Abstract

Motivated by an application in molecular biology, the prediction of biopolymer three-dimensional structures, an appropriate polymorphic tree search control structure has been implemented using a functional programming language to evaluate different tree search approaches to solve discrete combinatorial problems in three-dimensional space. The control structure is the basis of a constraint programming framework implemented in the functional programming paradigm. The non-strict semantic (lazy evaluation) and other features of higher-order functional programming languages have allowed to introduce constraint programming features in the functional programming paradigm.

1 Introduction

In the last two years, we have developed a constraint programming (CP) framework using the functional programming (FP) language Miranda[1] [1] to solve discrete combinatorial problems using tree search strategies. Higher-order FP allows for the expression and abstraction of algorithms and data structures. The CP framework is based on an abstract tree search control structure which has two functional parameters to control the domains before and during execution: a domain generating and a domain ordering function. The task of the programmer is to express his constraint satisfaction problem (CSP) by defining these two functional parameters which completely determine the behavior and efficiency of the search [2, 3].

[1] *Miranda* is a registered trademark of Research Software Ltd.

The use of this CP framework was demonstrated with a fundamental application in molecular biology: the prediction of biopolymer three-dimensional (3-D) structures. First the CP framework was tested on solving the 8-queen problem and a simplified version of the biopolymer 3-D structure prediction. Then, the Miranda prototype has been used to reproduce small biopolymer motifs [4, 5], demonstrating the feasibility of the approach for real problems. The prototype has recently been translated into C++ in order to solve larger problems.

2 Modeling biopolymers

A biopolymer is a sequence of molecules, called residues, linked by strong chemical bonds. Such sequences fold in 3-D space to adopt a minimum energy state, called native structure. A consensus has been established around the notion that the folding pathway is encoded in the sequence of residues. The native structure is necessary for biological function. The problem of biopolymer structure prediction has been much studied in the past 30 years but we still do not know how a sequence folds in 3-D space (see [6] for protein and [7] for nucleic acid structures). This problem is important because it is believed that knowledge about 3-D structure of biopolymers could help better understand their biological role. Furthermore, current sequence databases contain thousands of biopolymer sequences awaiting for their structure to be determined.

The most accurate method for determining the 3-D structure of a biopolymer is X-ray crystallography, a costly process that does not apply to all biopolymers. X-ray crystallography has been successful to determine the 3-D structure of only about a hundred biopolymers. Another method, Nuclear magnetic reasonance (nmr), which has been greatly improved over the past few years is particularly interesting because it allows for studies of molecules in solution. Under some conditions, nmr experiments could produce enough proton-proton distances (Nuclear Overhausser Effects (NOE) distance restraints) such that distance geometry algorithms [8] could be applied to produce accurate 3-D models. However, like X-ray crystallography nmr cannot be applied to all biopolymers. So far, nmr has only been applied successfully on small biopolymers.

The determination of biopolymer structures relies on the determination of structural data using theoretical and experimental methods, as well as, on computer programs that can transform structural data into 3-D models. Biopolymer structure modeling is possible with the identification of tertiary interactions occuring between pairs of residues which stabilize the native conformation. Some of these interactions define common structural motifs, called secondary structure. Theoretical methods are developed in order to infer secondary and tertiary structural information from sequence data. At present time though the accuracy and quantity of such infered information is not sufficient to accurately predict biopolymer 3-D structure.

Experimental methods are still the most productive source of biopolymer structural data. Proximity of atoms or residues under some conditions can be detected by ultraviolet irradiations, or evaluated by their participation in chemical reactions. The translation of this type of data into 3-D constraints is however more complicated since local modifications could alter, or even disrupt, the native structure. Nevertheless, although less precise than NOE distances, this type of information is extremely useful for 3-D modeling.

3 Variables, domains and constraints

Biopolymer modeling has been defined as a discrete combinatorial problem [4]. Solving this CSP consists of finding all biopolymer structures that satisfy available structural data. In this mapping, each residue is a variable. The values for each variable are taken into the cartesian product

between a set of residue conformations and a set of 3-D transformations. The set of conformations were determined by statistical analysis of all possible residue conformations. The transformations matrices can only be computed when all residues involve in a relation are appended to the structure. This introduces some complexity in the writing of the search algorithm if a language with strict semantic is used. A complete description of the domains prior to execution is required unless a more sophisticated mechanism is implemented. The non-strict semantic of higher-order FP languages allows for the description of such domains without any problem. The domains are not evaluated before all necessary residues are appended to the structure. In this way, the task of the programmer consists of simply describing the domains. The implementation of any control mechanism is not necessary.

Secondary structure information determines the possible conformations and transformations to be assigned to each residue. Tertiary information is introduced in a constraint function called by the domain generating function which is applied each time a new residue is appended to the structure. The order in which the residues are appended is determined by the user. The control structure performs a backtracking where the domains for each variable is computed dynamically. This search procedure is sound and complete. All models produced are consistent with the input data and all models described by the input data are produced. One of the advantage of this approach and of the use of a general function to introduce structural data is that all types of structural data produced from theoretical and experimental methods can be represented. Other approaches using penalty functions or distance matrices are restricted to distance constraints.

4 Scripts

A biopolymer modeling problem can be defined using a syntax especially developed for this application. All information are introduced in a script composed of three different sections: SEQUENCE, CONSTRAINT and GLOBAL. The SEQUENCE section allows for the introduction of the sequence and secondary structural information. For each residue, an identifier, a set of 3-D transformations, the set of residues which allows for the computation of the 3-D transformation, and a set of residue conformations are given. The CONSTRAINT section allows for the introduction of tertiary structural information. Finally, the GLOBAL section allows for the introduction of constraints that must be verified between every pair of residues, such as collisions.

Since there are many different scripts for the same set of data, the task of the modeler is to find a script that will produce solutions. The production of solutions is interpreted as a proof for the structural data or hypotheses. However, refutation of structural theorems is more difficult since one would have to generate all possible scripts. We are currently working on ways to automatize the generation of scripts. The script syntax we have developed could be thought of as an assembly language for biopolymer modeling.

5 Optimization and hypotheses

For many problems few structural data is available and too many models are produced. Nevertheless, modeling is possible by introducing hypothetical constraints infered from available ones and making use of heuristics to distinguish between several solutions. This type of information can be represented by an objective, or preference, function to be optimized during the search. The evaluation of partial solutions with the preference function and the use of constraints guide the search towards locally optimal structures that satisfy available data. Biopolymer structures of low energy that include tertiary contacts between non adjacent residues are often preferred

to structures that do not create tertiary contacts. These criteria are not sufficient to define a preference function that will localize global minima and an exhaustive search is still required to ensure soundness and completeness.

More recently, we have augmented the tree search control structure to introduce the preference function. In addition, partial solutions must be kept in a queue of partial solutions to be explored such as in a branch and bound algorithm. Using a simplified model for representing biopolymers, we are investigating different folding theories and other modeling protocols using this extended tree search control structure.

6 Conversion to C++

The translation of the control structure and domain generating function to C++ was somehow straight forward. In C++, the tree search procedure was implemented using an iteration control structure compared to the recursive function in the Miranda prototype. Many flag bits were necessary to control the evaluation of different variables in order to mimic lazy evaluation. On the same problem the C++ version is approximatively 50 times faster than the Miranda prototype on the same computer.

7 Conclusion

A CP framework implemented in a higher-order FP language, Miranda, has been used as a prototype to evaluate different search strategies for solving an important problem in molecular biology. The use of a domain generating function has simplified the writing of efficient programs since such a function is compatible to *a priori* pruning allowing for the incorporation of problem specific optimization.

Polymorphic types and lazy evaluation are crucial features for applications such as biopolymer structure prediction. Polymorphic types are necessary for the implementation of generic control structures. The implementation of the control structure was also greatly simplified using currying. Currying consists of defining functions from a partial instanciation of the parameters of another (more general) function.

Lazy evaluation allows for the description of domains that are only evaluated dynamically at proper time. Lazy evaluation allows for the description of very large, even infinite, domains efficiently since no evaluation is performed unless necessary. At some point in the execution of the algorithm, the domain could considerably be narrowed using problem specific information. Premature evaluation of the inital domains is inefficient.

The tree search control structure presented in this article could be introduced in the standard environment of functional languages to introduce CP features in the FP paradigm.

References

[1] D. A. Turner. Miranda — a non strict functional language with polymorphic types. In P. Jouannaud, editor, *Conference on Functional Programming and Computer Architecture, Lecture Notes in Computer Science #201*, pages 1–16. Springer-Verlag, 1985.

[2] F. Major, G. Lapalme, and R. Cedergren. Domain generating functions for solving constraint satisfaction problems. *J. Funct. Prog.*, 1(2):213–227, 1991.

[3] F. Major, G. Lapalme, and R. Cedergren. Dérivation d'une structure de contrôle de retour-arrière. In Christian Queinnec, editor, *Actes des journées JLFLA92: Avancées applicatives*, volume bigre 76-77, pages 202–219, 1992.

[4] F. Major, M. Turcotte, D. Gautheret, G. Lapalme, E. Fillion, and R. Cedergren. The combination of symbolic and numerical computation for three-dimensional modeling of RNA. *Science*, 253, September 1991.

[5] D. Gautheret, F. Major, and R. Cedergren. Modeling 3-d structure of RNA using discrete nucleotide conformational sets. Journal of Molecular Biology, 1993.

[6] Thomas E. Creighton. *Proteins, Structures and Molecular Properties*. W. H. Freeman and Company, 1984.

[7] W. Saenger. *Principles of Nucleic Acid Structure*. Springer-Verlag, New-York, 1984.

[8] L.M. Blumenthal. *Theory and applications of distance geometry*. Chelsea, Bronx, NY, 1970.

2lp: Linear Programming and Logic Programming

Ken McAloon and Carol Tretkoff
Logic Based Systems Lab
CUNY Graduate Center and Brooklyn College
2900 Bedford Avenue
Brooklyn, NY 11210
mcaloon@sci.brooklyn.cuny.edu
tretkoff@sci.brooklyn.cuny.edu

The 2lp system is a step in the "Operatica Program," a project whose grand design is to provide an elegant and powerful programming language environment for combining AI and OR methods for decision support software systems. The term "Operatica" was coined by J.L. Lassez to suggest an analogy with the Mathematica system which provides a programming environment for symbolic mathematical computation. In the dialogue between AI and OR, there are two basic themes: (1) declarative programming and the notion of logical consequence and, (2) procedural programming and the search algorithm in its many variations. Integrating AI and OR requires an environment that combines a modeling language with a logic based language. 2lp, which stands for "linear programming and logic programming," has the simplex based search mechanism of linear programming and the backtracking mechanism of logic programming built in. 2lp is both an algebraic modeling language and a logical control language. By bringing these techniques together in a language which has standard C style syntax and treats the mathematical module in an object-oriented way, this technology provides very powerful and usable tools for decision support programming.

The design decisions that led to 2lp were based on the following considerations:

- For run time efficiency the system should enforce a restriction to linear constraints at compile time.
- The array rather than the linked-list is the natural data structure for mathematical modeling.
- Communication between the logic and the numerical solver should be primitive to the language.
- A small language with standard syntax and with explicit integration of procedural programming constructs would be accessible to decision support programmers.
- The power of constraint programming is such that a compact language is sufficient for the intended applications.
- The system should be callable as a set of library routines.
- Hooks for expandability to an or-parallel system should be built in to the system.

In this paper, after discussing the relationship of 2lp to other programming paradigms and introducing some of the features of the language, we present some examples of 2lp applications and discuss future directions. Other applications can be found in [McAloon,Tretkoff 2], [McAloon,Tretkoff 3].

1. 2lp and other programming paradigms

2lp introduces a new built-in data type `continuous` in order to capture the modeling methods of linear programming. Constraints on these variables define a polyhedral set. The representation of this convex body includes a distinguished vertex, called the *witness point*. Other structural information and communication routines are also connected with the polyhedral set. This collection forms an "object" to use object-oriented programming (OOP) terminology. 2lp is a "little language" that supports these constructs and has its own interpreter. On the other hand, object-

oriented programming as in C++ enables one to extend a language with new data types without writing an interpreter for the expanded language. Because of the need for logic based control and because of the permissions and restrictions on the ways in which continuous variables can be used, the C++/OOP method can not be used for the purposes for which 2lp is designed; a little language approach proves necessary.

There has long been a debate in the AI world over the relative importance of declarative versus procedural programming. 2lp enables the programmer to make clean distinctions between the declarative and procedural aspects of a model and facilitates combining them for effective problem solving. The CLP languages Charme and CHIP integrate a declarative treatment of constraints with integrality and logical requirements in a larger programming framework and are an important development. On other hand, in CLP(R), in Sicstus Prolog with constraints etc., the programmer must provide the routines for driving continuous variables to integer values or to ranges that satisfy given logical requirements. The same is true of 2lp. In 2lp, this uncommitted approach is turned into a feature and tools are provided so that the programmer can tailor the problem solving method to the application at hand. These tools enable the modeler to use fuzzy logic and insight provided by the geometry of the polyhedral object defined by the constraints. This also means that the modeler can utilize powerful OR techniques that are used in MIP solvers such as Specially Ordered Sets of various types. In a similar vein, Prolog III provides built-in facilities which control search for an integer solution to constraints via communication with the state of the solver.

2lp is different from algebraic modeling languages such as AMPL [Fourer,Gay,Kernighan] in two important ways. First it is a logical control language as well as a modeling language. Second, 2lp supports parameter passing and procedure based modularity. In this it differs from current modeling languages in which the program basically consists of global variables and loops to generate constraints. On the other hand, AMPL can support network models directly and supports a set-theoretic view of a model as well as an algebraic view. AMPL also supports symbolic representation of indices and provides powerful and original logical/arithmetic connectives and quantifiers that operate at the level of the description of a constraint. Another feature of AMPL is the way it can enforce integrity constraints on the parameters of the model; the linkage between the data (concrete and large) and the model (abstract and small) is a priority in the AMPL system. Both 2lp and AMPL have a "symbolic computation" facility that recognizes that linear combinations of linear expressions are linear; this means that the modeler can employ natural mathematical expressions in constraints and is not forced to express constraints in some "canonical" form.

The 2lp language has evolved out a series of smaller languages which were developed step by step to maintain the best balance among the logic module of the language, the linear programming module and the intended applications. The decision to restrict the mathematical solver to linear constraints was based in part on the fact that the atomic operations of a programming language should be efficient. This was also motivated by the fact that the range of applications that can be handled with the combination of logic and linear constraints is remarkably rich. The theoretical basis for this work was developed in [Cox,McAloon,Tretkoff] which contains an analysis of the CLP scheme in terms of the computational complexity of the "halting problem" of languages which are instances of the scheme. The analysis also calibrates the role of the underlying logic used - propositional, relational, functional. Let D denote the ordered additive group of the real numbers as a Z-module. Then in the terminology of that paper, implementations of minimal-CLP(D) and conservative-CLP(D) were done before the current version of the language was designed and implemented. The minimal and conservative languages are based on propositional and relational logic respectively. The logic of full 2lp has a more complex character. All along an important consideration was the challenge of compiling this new kind of programming language building on the work of [Warren], [Jaffar,Michaylov] and others.

As a language for AI applications 2lp is dramatically different from LISP and Prolog. For classical AI applications, LISP and Prolog have certain definite strengths. Both support symbolic data and very flexible handling of types. Each is built on one mighty algorithm - lambda conversion in the case of LISP and unification in the case of Prolog - which brings a uniformity and declarative quality to programming [Lassez]. Both make programming with recursion very natural and recursion is the control mechanism of choice when working with data and data structures whose semantics are free in the mathematical sense. This is in contrast to classical imperative languages which are oriented toward numerical processing with its well-understood underlying mathematical semantics. Moreover, the remarkable transparent memory management of LISP and Prolog, made possible by the role of the linked-list as the basic data structure, makes writing code for various search strategies very elegant in these languages: the stacks and

queues are managed by the system and garbage is collected in the background. To this Prolog adds its built-in logical control and more declarative programming facility. The weak point, if you will, of these languages historically has been their handling of numerical computation. The development of the CLP languages was motivated to a large extent by the aim of integrating numerical computation naturally into the declarative framework of logic programming. These systems have been very successful.

Traditionally, the language of choice for heavily numerical computing is FORTRAN. Though built on a static picture of memory management based on the array, FORTRAN has been in fact used for many interesting systems which link AI and mathematics. The language C occupies a middle position and supports arrays and pointers which allow for an elegant treatment of linked-lists which captures much of the modeling power of LISP and Prolog. 2lp itself does not support pointers, and if queues, stacks or linked-lists are used in a 2lp program, they are coded in the FORTRAN style. However, such data management, especially if it is a computationally significant part of the application, is best done in C itself and handled by means of calls from 2lp to external C functions. The fact that 2lp follows the C picture of memory allocation, parameter passing and overall arithmetic syntax makes this integration of C functions very natural.

2. The 2lp language

The basic data types of 2lp are `int`, `double` and `continuous`, which is a new type for working with constraints. The data structures are arrays of elements of a given basic type. Arrays are the natural data structure for mathematical programming and facilitate working with constraints. In a 2lp program, variables of type `continuous` function in the declarative manner of the variables of linear algebra or linear programming. As a program progresses, linear constraints on the `continuous` variables are generated and define an evolving polyhedron in n-dimensional space, where n is the number of `continuous` variables declared in the program.

While the fundamental operation on variables of type `int` and `double` is assignment the `continuous` variables in a 2lp program are constrained by linear equalities and inequalities. Thus 2lp extends to this new type the arithmetic operators +,- and * (for coefficients) and the relational operators ==,<= and >=. The = operator is used for assignment as in C. The operators !=,< and > are also extended to continuous variables and are interpreted by means of negation-as-failure. By default `continuous` variables are constrained to be non-negative - unless a negative lower bound is explicitly imposed. A program consists of declarations of global storage including external storage and external functions, a main procedure called `2lp_main()` and other procedures. All storage locations created in the program are either global variables, external global variables or are declared in `2lp_main()`; the exception is that loop control variables are introduced locally. Procedures take parameters by value or by reference following C and C++ conventions. Loop variables can only be passed by value; `continuous` variables can only be passed by reference. Variables which are passed by value to a procedure can not be assigned in that procedure. As in logic programming, procedures may contain disjunctive control constructs which generate "choice points" in the computation. These are nodes in the logic programming search to which the program can return later in its run. The system uses depth-first search and chronological backtracking. Upon "failure" the backtracking mechanism resets the constraints and the continuous variables to their state at the previous choice point; loop variables and variables passed by value are also reset to their values at the choice point. The state of the global variables of type `int` and `double` and of the variables of these types passed by reference is left unchanged by backtracking. In classical logic programming, the "logical variable" is not assigned values but is monotonically constrained by unification. Thus the `continuous` variables, the loop variables and the variables passed by value capture the role of the "logical variable" and provide the system its declarative kernel.

As a programming language, 2lp is small; its ambitions are circumscribed and focused. It is designed to be embedded in the larger programming scheme of things and to be used as a set of library routines. Arrays of `doubles` and `ints` and C-functions can be provided by the principal system and addressed as `extern` by the 2lp model. The language is built on an abstract machine model called the S-CLAM which is an analog of the WAM and CLAM in the smaller 2lp context [McAloon,Tretkoff 1]. For the record, the S-CLAM supports tail recursion and classical logic programming formats such as multiple procedure definitions and cut. The 2lp architecture provides for a very modu-

lar linkage between the logical control and the mathematical constraint solver. As a result, 2lp supports use of optimization libraries such as Cplex and OSL as well as its own simplex based mathematical code. This means that 2lp can be used for applications that require state-of-the-art optimization software and the system can move with changes in the mathematical programming world. (It is also to be hoped that by interacting with systems like 2lp, optimization software will become more responsive than at present to requirements like incremental loading and deleting of constraints.)

2lp provides language tools for programming the desired semantics for an optimization application. For a classical linear program the semantics of optimization are straightforward. When a richer mix of programming constructs and logic is introduced, there are many possible interpretations to "max: ... subject_to ... ." In particular, the programmer will want to control the state of the geometric object defined by the constraints both during and upon exiting this block of code. For that reason, 2lp supports control mechanisms and some built in optimization constructs for writing code that expresses the semantics intended by the modeler. This is especially helpful when optimization routines occur as subroutines in a larger application.

The logic programming "negation-as-failure" is supported by 2lp. It turns out that in programming with constraints, it is double-negation-as-failure that is a most powerful and useful tool. The reason for this is simple: a call to `!!add_constraints()` is a test for the consistency of the constraints generated by a call to `add_constraints()` with the current geometric object; this test for consistency leaves the object unchanged. It is this type of query that is needed for Land and Doig search, for A* search, for computing heuristics, for lookaheads etc. Again for the record, 2lp interprets the open comparison operators <,>, and != by means of negation-as-failure. This is a coherent reading which preserves the closed, convex set interpretation of constraints and which gives these open operators a natural role to be used as guards or checks.

2lp can be used as a stand-alone system or as a callable library. In stand-alone mode, a 2lp model is sent directly to the compiler/interpreter. In library mode, 2lp is called from C as a function; the 2lp function is sent the whereabouts of the data and C functions it is to use as well as the 2lp model; the model can be sent either as a file or as a pre-compiled C data structure.

3. Dynamic alternation

The 2lp system takes a structured approach to logical modeling. It supports `and`-loops, `or`-loops and other logical constructs. This has the advantage of making the structure of the logic of a program clear and high-level. Simply put, it makes the alternating structure of a logic program easier to grasp. In classical logic programming languages, logical loops must be expressed by means of recursion using both multiple procedure definitions and cut. In 2lp logical loops and logical alternation can be expressed in a structured way. By way of example the following 2lp code uses an `or` loop nested in an `and` loop to generate permutations of `N` symbols in lexicographical order. It is a variation of the classical algorithm given, for example, in [Reingold *et al.*]. The `and-or` construct can often be used to enumerate the possibilities for a search problem. It is the `and` that moves the search forward to the next level. When a failure occurs, the `or` has kept track of where to start trying next. The loop control variables are handled declaratively in 2lp in that their proper values are restored upon backtracking. In the first bit of code below, failure is triggered by a `fail` statement, but in the scheduling example that follows, it is when a potential schedule cannot satisfy the constraints, that failure occurs and the next permutation is tried. This method of generating permutations is useful in 2lp because the lexicographic order meshes correctly with backtracking mechanism of the logic programming search.

```
#define N ...
int permu[N];
int p[N][N];
2lp_main()
{
      and(int i=0;i<N;i++) p[0][i] = i; //The identity permutation

      and(int i=0;i<N;i++)
```

```
        or(int j=0;j<N-i;j++){
              if j == 0; then and(int k=i+1;k<N;k++) p[i+1][k] = p[i][k];
              permu[i] = p[i][i+j];
              if j != 0; then p[i+1][i+j] = p[i][i+j-1];
        }
    and(int i=0;i<N;i++) printf("%d ",permu[i]);printf("\n");
    fail; //forces backtracking to generate next permutation
}
```

The **and-or** construction is natural to use as a form of ∀∃. In this way, in working with NP-problems, it can play the role of the non-deterministic guess in the Garey-Johnson model [Garey,Johnson]. In the context of Complexity Theory, the non-determinism is factored out as one sequence of guesses; in the context of actual programming, candidate guesses are generated incrementally and so take an **and-or** or ∀∃ structure.

In 2lp procedural techniques for doing recursion and for restoring state in a search application are often used in conjunction with the built-in logic programming machinery which restores continuous variables and loop control variables. Let us look an example which uses the lexicographic ordering on permutations. Schedules which require permuting a sequence of tasks in order to meet delivery, production and other kinds of constraints are called *permutation schedules*. The following code segment is taken from an application to a permutation scheduling problem. The continuous variable `start_time[i]` represents the starting time of task `i`, `permu[k]` is the task slated for the `k`th position in the schedule, `prod[i]` is the production time required for task `i` and `dt[i]` is the time by which task `i` must be finished.

```
#define N ... // number of tasks to schedule
extern double prod[], //prod[i] is time required for task i
              dt[]; // dt[i] is delivery time for task i
int permu[N];
int p[N][N];
continuous start_time[N];
           //start_time[i] is the start time for the ith task

2lp_main(){
     //declarative knowledge
     //priority constraints such as
     start_time[i] >= start_time[j] + prod[j]; //task i depends on task j
        ...
     //delivery constraints such as
     start_time[i] + prod[i] <= dt[i]; //task i must be finished by time dt[i]
        ...
     //throughput constraints such as average job must be finished
     //within 2/3 of the total scheduled time
     (1.0/N)*sigma(int i=0;i<N;i++)(start_time[i] + prod[i]) <= .
              .66 * sigma(int i=0;i<N;i++) prod[i];
        ...
     //procedural part of program
     and(int i=0;i<N;i++) p[0][i] = i;//initialize identity permutation
     if schedule_all();
     then and(int i=0;i<N;i++)
          printf("Schedule task %d in the %dth position\n",permu[i],i);
     else printf("There is no feasible schedule\n");
}
```

```
schedule_all() {
      and(int i=0;i<N;i++)
            or(int j=0;j<N-i;j++) {
                  if j == 0; then and(int k=i+1;k<N;k++) p[i+1][k] = p[i][k];
                  //schedule task p[i][i+j]
                  start_time[p[i][i+j]] ==
                        sigma(int k=0;k<i;k++) prod[permu[k]];
                        // fails if inconsistent with setup constraints
                        // and those generated thus far by this statement
                  permu[i] = p[i][i+j];
                  if j != 0; then p[i+1][i+j] = p[i][i+j-1];
            }
}
```

4. Factoring the logic in and out

As a programming language for classical MIP applications, the fact that 2lp is a logic based language is a significant advantage. For it can handle logical decisions either by means of additional 0-1 variables or by direct encoding of the logic using the 2lp program constructs. Thus, for instance, 2lp can make full use of an important idea that comes from MIP modeling, namely the use of continuous variables as fuzzy logical variables. For example, if the program contains several "constants" such as a fixed-cost or capacity whose values are in fact functions of a single logical decision, then these values can be parametrized as (linear) functions of a continuous variable; this variable is then bound by 0 and 1 and is made equal to 0 or 1 depending on the logical decision made. This yields a fuzzy or continuous declarative approximation to the model sought and can be very useful both from a heuristic point of view and a pruning point of view. However, MIP modeling is not based on a logic language and logical disjunction and other connectives can only be expressed by the introduction of additional 0-1 variables. These variables are used to encode switches that express the logical relationships defined by the application. On the negative side, the introduction of these variables can greatly swell the total number of variables of the model and can add a considerable number of constraints. (This means adding rows and columns to the matrix sent to the linear programming solver; this is exacerbated by the fact that a row also will add an additional column for its slack variable. This comes down to a quadratic increase in the size of this matrix.) With the logical control that is built in to 2lp, coding logic in terms of 0-1 variables can most often be eliminated; the result can be both elegance in modeling and speed in performance. However, in other cases, additional variables can serve to "tighten" a model and to make the "linear relaxation" fit the disjunctive model more closely. This closeness of fit can be most important in the phase of an optimization program where the job remaining is to verify that the current best solution cannot be improved. 2lp facilitates the development of both loose and tight models and the transition back and forth between them. For examples, we refer the reader to [McAloon,Tretkoff 3].

5. An Example: Goal Programming with Logic

The term *goal programming* is used to describe situations where there are several criteria that must be taken into account to have the best solution to a problem. Goal programming is usually done only for linear programming models. In the application that follows we consider a goal programming case where logic, in the form of integrality requirements, is needed in addition to the straight linear model.

In this application the task is to schedule toll takers. An analysis of traffic patterns has determined the number of employees required for each hourly period during the 24 hour day. Each worker comes on for a 9 hour shift with a 1 hour break in the middle. Management wants to minimize the total number of workers assigned each day. Thus the program must determine the number of workers who should start their shifts at each hour of the day so as to minimize the total number of workers hired. The model requires, of course, a solution which determines an integral number of workers to start at each hour, and it is this requirement that necessitates a logical search strategy. Unavoidably, there

will be periods where the number of workers available exceeds the number required for that period. These people will of course be able to take care of other duties beside collecting tolls. It is desirable to find a schedule which distributes these extra workers as evenly as possible during the day so that continuity can be maintained in these auxiliary jobs.

The first task of the model will be to determine the minimum number of workers required. This will be done with a first objective function. Then, once the minimal total number of workers is found, the next task is to distribute the workers who will not be collecting tolls as evenly as possible during the 24 hour day. For that two stratagems will be employed in succession. First, the largest number of extra workers that can occur at any one hour will be minimized and then the number of different time periods that can have extra workers will be maximized. To do all this, for the first two phases of the program, i.e., determining the minimal number of workers needed and then minimizing the largest number of extra workers at any hour, the built-in optimizer construct `min: ... provided {}` is used; with this construct, after the optimum is determined the objective function for that optimization call is set equal to its optimal value but otherwise the continuous variables of the program are only constrained by the constraints that were active before the optimizing call was made. This construct supports a "bluff": since the number of workers is necessarily an integer, to avoid alternative solutions which determine the same number of workers, each successive solution found during a search phase of the problem should be required to better the previous solution by at least 1. The command `bluff(1.0)` sets this machinery in motion. (The term "cheat" is also used in this context in the literature, and "bluff" is sometimes used to denote an initial upper bound on the objective function in a minimization problem or lower bound in a maximization problem.) For the final task, the `max: ... subject_to {}` construct is used; upon completion of this block the continuous variables are fixed to the values the witness point had when the optimal solution was found.

The only data in the model is the number `d[j]` of toll takers required for each hourly period.; this data is obtained as an external array of integers. The only constraints needed are the ones that sum up the number of toll takers on duty for each hourly period `j`. We will let the continuous variable `x[j]` represent the number of workers who come on at the `j`th hour. Looking ahead to hour `j+8`, the workers on duty will be all those who came on at some time `j+i` for `i=0,...,3` and `i=5,...,8`. For each `j+8` this requirement can be expressed as

```
sigma(int i=j;i<=j+8;i++)x[i%N] - x[(j+4)%N] <= d[(j+8)%N]
```

where `N` is a defined constant equal to 24. Naturally, we have to subtract the term `x[(j+4)%N]` which represents the workers on break. If we let `sur[i]` be the number of surplus workers at hour `i`, a variable which will be used in the second and third phases of the solution process, this can be written

```
sigma(int i=j;i<=j+8;i++)x[i%N] - x[(j+4)%N] - sur[(j+8)%N] == d[(j+8)%N]
```

In each of the three phases of the program, in order to meet the integrality requirements on the `x[i]`, the model employs a "lazy" strategy which is akin to a priority argument in Recursion Theory. The linear relaxation is optimized and there is a circular loop around the 24 time periods. If `wp(x[i])`, the `x[i]`-coordinate of the witness point, is currently at an integer value, this time slot is skipped over and the loop continues until a variable is encountered such that the witness point's coordinate at that variable is not integral; then the process is begun to drive that variable to an integral value. To describe this process, let `t` be equal to `floor(wp(x[i]))`; then the search loops through the constraints `t == x[i]`, `t-1 == x[i]`,... continuing upon backtracking toward `0 == x[i]`. A fact about convexity is brought into play - if fixing `x[i]` at a an integer value `t-k` is linearly infeasible, then the search in that direction can be cut off since no smaller value will be linearly feasible. After the search in the downward direction, the search proceeds upward from `t+1`. Naturally, a variable which was at an integer position might be "injured" in the branching and optimization process required to make another variable have an integral value. Either this variable will return to an integral position or it will eventually be reached in the circular loop and branched upon. Thus all integrality "requirements" are met.

```
#define N 24 // hours in the day
#define M 8 // 8 working hours

continuous x[N]; // x[i] is number of workers starting at hour i
continuous sur[N]; // sur[i] is number of surplus workers at hour i
```

```
continuous z,u, ex[N],tra[N]; // auxiliary variables

int s; // circular loop counter
extern double d[]; // data entered in external C code

2lp_main()
{
      sigma(int i=0;i<N;i++) x[i] == z; //z is to be minimized

      set_up_constraints();

      schedule1();
      schedule2();
      schedule3();

printf("The total number of workers required is %d\n",nint(z));
printf("There will be surplus workers during %d hourly periods\n\n",nint(sig-
ma(int k=0;k<N;k++) tra[k]));
printf("The schedule recommended is \n\n");
and(int i = 0;i<N;i++)
      printf("At hour %i \t bring on %d workers\n",i,nint(x[i]));
}

schedule1(){ // determine minimum number of toll takers required
      min: z; provided {//only restores optimal value of objective function
            bluff(1.0);
            optimize();
            s = N-1;
            lazy_loop();
      }
}

schedule2(){ // minimize the maximum surplus workers
      and(i=0;i<N;i++) sur[i] <= u;
      min: u; provided {
            bluff(1.0);
            optimize();
            s = N-1;
            lazy_loop();// find minimum hourly surplus bound
      }
}

schedule3(){ // distribute surplus workers more broadly

//cap on number of surplus workers at each hour
      and(i=0;i<N;i++) sur[i] <= u;

//tra[i] is "characteristic function" for presence of surplus workers
      and(i=0;i<N;i++) tra[i] <= 1;

// decompose hourly surplus
      and(int k=0;k<N;k++) sur[k] == ex[k] + tra[k];

// favor wide distribution of surplus workers
```

```
      max: sigma(int k=0;k<N;k++) tra[k]; subject_to {
                  bluff(1.0);
                  optimize();
                  s = N-1;
                  lazy_loop();
      } //subject_to restores value at optimum of
        //all continuous variables

}


lazy_loop(){
      and(int i=0;i<N;i=(i+1)%N)
            if !integral(x[i]); // fabs(wp(x[i]) - nint(x[i])) < EPSILON
                  then down_and_up(i,floor(x[i]),x[i]);
            else if i == s; then break; //break exits and-loop with success
}

down_and_up(int i,double t,continuous x){//i,t passed by value, x by reference
      either // down
            or(int k=t;k>=0;k--)
                  if x == k; then {optimize(); s = i;}
                  else break;
      or // up
            or(int k=t+1;k<=M;k++)
                  if x == k; then {optimize(); s = i;}
                  else break; //break exits or-loop with fail
}

set_up_constraints(){

      and(int j=0;j<N;j++)
            sigma(int i=j;i<=j+M;i++) x[i%N] - x[(j+4)%N] //9 hours less break
                        - sur[(j+M)%N] == d[(j+M)%N];
}
```

6. Future directions

An interesting topic combining logic and constraints is propositional theorem proving. Experimental work in this field using polyhedral methods is being carried out in several research centers in the USA and abroad, e.g. [Hooker],[Comon *et al.*], [Hahnle], [Bell *et al.*]. Our starting point is the analysis of [Blair,Jeroslow,Lowe] and Jeroslow's watershed monograph [Jeroslow] where it is argued that Davis-Putnam-Loveland (DPL) techniques are more amenable to constraint methods than resolution based techniques. The relation of propositional consequence is co-NP complete. With resolution methods, a problem to show there is no polynomially long path of a certain type is transformed into an existential search for a successful path of possibly exponential length. The proper analogy from Recursion Theory is the equivalence of Π^1_1 sets of integers and ω_1^{CK} -recursive sets of integers. (Note that in this view NP is analogous to co-RE and the analytic sets of recursion theory and descriptive set theory.) The DPL approach keeps the consequence relation in its co-NP form and makes it akin to a MIP optimization problem. We have implemented a version of DPL in C calling 2lp to handle the constraints. The first difference with symbolic DPL methods is that the unit resolution check for consistency is done automatically by the simplex based linear program-

ming solver. Note that this also means that the linear programming solver provides a complete theorem prover for the Horn clause fragment of the set of clauses. Using the 2lp programming language tool we will also be able to add to the continuous mathematical version of DPL heuristics and search techniques which exploit the geometry of the polyhedral set defined by the propositions interpreted as constraints. This puts at our disposal, heuristics such as the Balas-Martin heuristic from Integer Programming, greedy and Gray code methods, the Jeroslow heuristic for DPL as well as some new opportunistic methods for driving continuous variables to discrete values that we have developed in our own work.

Among important non-traditional connectives are generalized disjunctions such as 5 choose 3. When expressed in clause form these connectives are not very amenable to unit resolution and similar symbolic techniques. However, it can be shown that in conjunction with constraint methods, these generalized disjunctions can be used to detect early failure in the search for a satisfying assignment to a set of propositions. Let us look at a concrete example. The generalized disjunction "At most one of p,q,r is true" transforms into three clauses, namely, {p',q'},{q',r'},{p',r'}. If we add to these the clauses {p,q},{p',q},{p,q'},{q,r} the resulting set is inconsistent. However, unit resolution cannot detect this inconsistency and the symbolic DPL algorithm can not detect it without resorting to the splitting rule. As linear constraints, these clauses become

$$q + r >= 1;\ (1-p) + (1-q) >= 1;\ (1-q) + (1-r) >= 1;\ (1-p) + (1-r) >= 1;$$

$$p + q >= 1;\ (1-p) + q >= 1;\ p + (1-q) >= 1;\ 0 <= p,q,r <= 1;$$

This set of constraints is still consistent as a set of linear constraints; in fact, by Chapter 5 of [Jeroslow], a set of clauses with no unit clauses is always consistent from the linear point of view. But, if we also add the generalized disjunction "At most one of p,q,r is true" in the form $p + q + r <= 1$, the linear constraints become inconsistent.

2lp has been designed with the purpose of bringing into software control and search mechanisms which in MIP and other constraint systems are either hidden in a black-box or simply not available. This aim is being realized to a definite extent and promises to continue to extend its range. The next major challenge is to bring into the 2lp orbit (or that of another constraint language) the techniques of "polyhedral theory" and "branch 'n' cut." Certainly the work on MINTO [Savelsbergh,Sigismondi,Nemhauser] makes one believe that this can be achieved with some success. In our opinion this will prove important both for optimization applications and constraint-based reasoning systems and theorem-provers. The reason is that "branch 'n' cut" methods encode a kind of "lemma" in the form of constraints which must hold at any solution below the current node in the search. Adding the constraint description of "At most one of p,q,r is true" to the clause description in the example above is a "lemma" in a similar way. These "lemmas" are special in that unlike an ordinary lemma, they become "resident" and stay in force for the rest of the computation once they are added to the constraint set. In traditional automated reasoning systems, in rewrite-based symbolic computation systems, and in ordinary human mathematical theorem proving, a search process must be invoked to find a previously established lemma and to apply it locally. The "resident lemmas" can be applied once and for all with no additional invocation required. As far we know, this technique has not been applied in expert systems or symbolic computation systems.

Let us make a "remark" in the mathematical sense: for simplicity, consider the situation where a model has no solution. (This is the case of an optimization model once the best possible solution has been found and it is the situation in a DPL based propositional theorem prover if the theorem to be proved is true!) For simplicity also suppose that a state or node in the search can only be reached one way in either depth-first or breadth-first search and finally suppose that the total number of nodes is finite. Then (it can be seen) both depth-first and breadth-first search visit all possible nodes. It follows from this remark that for all practical purposes in the above situation, exhaustive enumeration by depth first search is the most efficient means of verifying that there is no better solution among classical methods such as A*, Land and Doig etc. This is not a particularly pleasant prospect since it is so limiting; "polyhedral theory" and "branch 'n' cut" are the best candidates to look to at this time for some improvement. Parallel processing is another candidate; in [Atay] arguments are given as to why a parallel search will not visit unnecessary or duplicate nodes in this situation.

Another promising application of constraint programming is to respond to the challenge launched by [Dhar,-Ranganathan] where it is argued that rule based expert systems are superior to mathematical programming methods for attacking complex constraint problems; in brief they contend that MIP methods do not allow for local control of decisions and do not provide a mechanism for providing good partial solutions.

7. Concluding remarks

The 2lp system currently runs on UNIX workstations and 386/486 PCs. A parallel implementation which basically is an or-parallelization of the depth-first 2lp stack mechanism has been done. This work is reported on in [Atay,McAloon,Tretkoff].

The authors would like to acknowledge the support of NSF grant CCR-9115603.

8. Bibliography

[Atay] C. Atay, Parallelization of the Constraint Logic Programming Language 2lp, Ph.D. Thesis, City University of New York, June 1992

[Atay,McAloon,Tretkoff] C. Atay, K. McAloon and C. Tretkoff, 2lp: a highly parallel constraint logic programming language, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, March 1993

[Bell *et al.*] C. Bell, A. Nerode, R. Ng and V. S. Subrahmanian, Implementing deductive databases by linear programming, University of Maryland, Computer Science Technical Report, CS-TR-2747, UMIACS-TR-91-122.

[Blair,Jeroslow,Lowe] C. W. Blair, R. G. Jeroslow, J. K. Lowe, Some results and experiments on programming techniques for propositional logic, *Computer and Operations Research* 13(1986) 633-645.

[Colmerauer] A. Colmerauer, An introduction to Prolog III, *CACM* 33 No. 7 (1990) 69-91.

[Comon *et al.*] H. Comon, H. Ganzinger, C. Kirchner, H. Kirchner, J.-L. Lassez, G. Smolka (editors): Theorem Proving and Logic Programming with Constraints, Dagstuhl-Seminar-Report; 24 (9143)

[Cox,McAloon,Tretkoff] J. Cox, K. McAloon, C. Tretkoff, Computational complexity and constraint logic programming, *Annals of Mathematics and Artificial Intelligence*, 5(1992) 163-190.

[Dhar,Ranganathan] V. Dhar and N. Ranganathan, Integer Programming vs. Expert Systems: an experimental comparison, *Communications of the ACM* 33, No. 3 (1990) 323-337.

[Dincbas et al.] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, F. Berthier, The Constraint Logic Programming Language CHIP, *Proceedings of the International Conference on Fifth Generation Computing Systems*, 1988.

[Fourer,Gay,Kernighan] R. Fourer, D. Gay and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, 1993

[Garey,Johnson] M. Garey and D. Johnson, *Computers and Intractability*, W.H. Freeman 1979

[Hahnle] R. Hahnle, A new translation for deduction into integer programming, submitted for publication.

[Hooker] J.N. Hooker, Jr., Input proofs and rank one cutting planes, *ORSA Journal on Computing* 1, No. 3(1989) 137-145.

[Jaffar,Michaylov] J. Jaffar, S. Michaylov, Methodology and Implementation of a CLP System, *Proceedings of the 1987 International Logic Programming Conference*, edited by J.-L. Lassez, MIT Press, 1987.

[Jeroslow] R. G. Jeroslow, Logic-Based Decision Support, Mixed Integer Model Formulation, North-Holland, New York, 1989.

[Jeroslow,Wang] R. G. Jeroslow, J. Wang, Dynamic programming, integral polyhedra and Horn clause knowledge bases, *ORSA Journal of Computing* 1, No. 1(1989) 7-19.

[Lassez] J.-L. Lassez, From logic programming to lazy programming, manuscript.

[McAloon, Tretkoff 1] K. McAloon and C. Tretkoff, Subrecursive constraint logic programming, *Proceedings of the NACLP 1990 Workshop on Logic Programming Architectures and Implementation*, edited by J. Mills.

[McAloon,Tretkoff 2] K. McAloon and C. Tretkoff, *Logic and Optimization*, in preparation

[McAloon,Tretkoff 3] K. McAloon and C. Tretkoff, AI/OR Modeling in 2lp, Brooklyn College Computer Science Technical Report 93-4.

[Reingold *et al.*] E. Reingold, J. Nievergeld, and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, In., Englewood Cliffs, NJ, 1977

[Savelsbergh,Sigismondi,Nemhauser] M. Savelsbergh, G. Sigismondi and G. Nemhauser, Functional description of MINTO, a MIxed INTeger Optimizer, Georgia Institute of Technology Tech Report 30332, May 1992.

[Warren] D.H.D. Warren, An Abstract Prolog Instruction Set, Technical Report 309, SRI International, 1983.

An Incremental Hierarchical Constraint Solver

Francisco Menezes (fm@fct.unl.pt)
Pedro Barahona (pb@fct.unl.pt)
Departamento de Informática, Universidade Nova de Lisboa
2825 Monte da Caparica, PORTUGAL

Philippe Codognet (codognet@minos.inria.fr)
INRIA-Rocquencourt
BP 105, 78153 Le Chesnay, FRANCE

Abstract

This paper presents an incremental method to solve hierarchies of constraints over finite domains, which borrows techniques developed in intelligent backtracking, and finds *locally-predicate-better* solutions. A prototype implementation of this method, IHCS, was written in C and can be integrated with different programming environments. In particular, with Prolog producing an instance of an HCLP language. Possible applications of IHCS are briefly illustrated with a time-tabling and a set covering problem. Because of its portability and incremental nature, IHCS is well suited for reactive systems, allowing the interactive introduction and removal of preferred constraints [1].

1 Introduction

Modelling a real life problem by means of an explicit set of constraints always involve, to some extent, the abstraction of the context in which these constraints are to be considered. Decision support systems are usually concerned with supplying the decision maker with a set of alternative scenarios in which the most important constraints are satisfied [1]. Because not all constraints can be met by a solution, one might simply specify a set of mandatory constraints and select one of the possible solutions that satisfy such constraints.

Nevertheless, the choice of a good solution often depends on preferred criteria (forming the contextual background knowledge), not explicit in this approach to problem formulation. A more powerful approach consists of specifying all intended constraints in some hierarchy, i.e. qualifying them either as mandatory or as mere preferences, possibly with some associated preference strenght. In [2], a general scheme is proposed for Hierarchical Constraint Logic Programming (HCLP) languages, parameterized by $\mathcal{D}$, the domain of the constraints and by $\mathcal{C}$, a comparator of possible solutions.

The Incremental Hierarchical Constraint Solver (IHCS) that we have developed is intended as the kernel of a HCLP($\mathcal{FD}, \mathcal{LPB}$) instance of this scheme, where $\mathcal{FD}$ stand for finite domains and $\mathcal{LPB}$ is the *locally-predicate-better* comparator. Operationally, our approach diverges from the one presented in [2] because it is incremental. Instead of delaying the non-required constraints until the complete reduction of a goal, IHCS tries, in its forward phase, to satisfy constraints as soon as they appear. In case of inconsistency, a special backward algorithm is evoked.

This can be seen as an "optimistic" treatment of preferred constraints (i.e. we bet they will participate in the search for a solution), as opposed to the "pessimistic" view of [2] where non-required constraints (source of possible inconsistency) are delayed as long as possible. The advantage is to actively use these constraints for pruning the search space. This approach nevertheless requires a specialized backward phase where dependencies between constraints, caused by their handling of common variables, are exploited to identify pertinent causes of failure. This is done much in the same way as in intelligent backtracking [9] [4], although instead of finding pertinent choice points, IHCS identifies pertinent constraints to be relaxed.

Because of its portability and incremental nature, IHCS is well suited for reactive systems requiring constraint facilities, allowing the interactive introduction and removal of preferred constraints to further refine any solution found.

This paper presents a formal specification of IHCS and describes the algorithms that perform these transitions. The paper is organized as follows. Section 2 presents the formal specification of the basic IHCS,

[1] This work was developed at INRIA/Rocquencourt and at the AI Centre of UNINOVA and was funded by Délégation aux Affaires Internationales (DAI) and Junta Nacional de Investigação Científica e Tecnológica (JNICT).

as a set of transition rules over hierarchy configurations, together with supporting definitions. Section 3 describes the algorithms that perform these transitions. Some extensions to the basic IHCS are addressed in Section 4, to deal with the search for alternative solutions, the incremental removal of constraints and to cope with disjunctions of constraints. A set-covering application and a time-tabling application are presented in Section 5 and the conclusions are presented in Section 6.

2 An Incremental Hierarchical Constraint Solver - IHCS

A *constraint hierarchy* $\mathcal{H}$ is a set of *labelled constraints* c@*level* relating a set of variables ranging over finite domains. c is a constraint on some variables and *level* the strength of c in the hierarchy. Level 0 corresponds to the required constraints and the other levels to the non-required (or preferred) constraints. The higher the level, the weaker a constraint is.

A *valuation* to a constraint hierarchy $\mathcal{H}$, is a mapping of the free variables in $\mathcal{H}$ to elements in their respective domains, that satisfies all the required constraints (level 0). Given two valuations θ and σ, θ is *locally-predicate-better* than σ [2] if a) θ and σ both satisfy exactly the same number of constraints in each level until some level k, and b) in level $k+1$ θ satisfies more constraints than σ.

Given a constraint hierarchy $\mathcal{H}$ with n variables and m constraints, $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ denotes the set of variables and $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$ the set of constraints. In our notation, c or c_i designates any constraint from $\mathcal{C}$. The index indicates the *introduction order* of the constraint in the hierarchy.

Definition 1 (Constraint Store) A *constraint store* S is a set of constraints ordered by *introduction order*, i.e., if c_i and c_j belong to S and $i < j$ then c_i precedes c_j in S. Any operation on constraint stores preserves this ordering.

Definition 2 (Configuration) A *configuration* Φ of hierarchy $\mathcal{H}$ is a triple of disjoint constraint stores $\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \mathcal{US} \rangle$, such that $\mathcal{AS} \cup \mathcal{RS} \cup \mathcal{US} = \mathcal{C}$. $\mathcal{AS}$ is the *Active Store*, $\mathcal{RS}$ the *Relaxed Store* and $\mathcal{US}$ the *Unexplored Store*.

A configuration may be seen as a state of the evaluation of a hierarchy where the active store contains all the active constraints (i.e. those that might have reduced some domains of its variables), the relaxed store is composed by the relaxed constraints and the unexplored store is the set of candidates "queuing" for activation. We will denote that a store S is consistent by $S \not\vdash_X \bot$, where X designates a network consistency algorithm (e.g. X = AC for Arc-Consistency [6]). S_i denotes the subset of S containing only constraints of level i. A store $S' = S \cup \{c\}$ may be expressed by $c.S$ if c is the element of S' with lower introduction order or by $S.c$ if c is the element of S' with higher introduction order.

Definition 3 (Final Configuration) A *configuration* $\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \mathcal{US} \rangle$ of hierarchy $\mathcal{H}$ is a *final configuration* if, given the initial domains of the variables the following conditions hold:

1. $\mathcal{AS} \not\vdash_X \bot$;
2. $\mathcal{AS} \cup \{c\} \vdash_X \bot \quad (\forall c \in \mathcal{RS})$;
3. $\mathcal{US} = \emptyset$.

Definition 4 (Locally-Predicate-Better) $\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \mathcal{US} \rangle$ is *locally-predicate-better* than $\langle \mathcal{AS}' \bullet \mathcal{RS}' \bullet \mathcal{US}' \rangle$, if and only if exists some level $k > 0$ such that:

1. $\#(\mathcal{AS}_i \cup \mathcal{US}_i) = \#(\mathcal{AS}'_i \cup \mathcal{US}'_i) \quad (\forall i < k)$;
2. $\#(\mathcal{AS}_k \cup \mathcal{US}_k) > \#(\mathcal{AS}'_k \cup \mathcal{US}'_k)$.

Definition 5 (Best Configuration) A final configuration Φ is a *best configuration* if there is no other final configuration Φ' which is locally-predicate-better than Φ.

Definition 6 (Promising Configuration) $\Phi = \langle \mathcal{AS} \bullet \mathcal{RS} \bullet \mathcal{US} \rangle$ is a *promising configuration*, denoted PC(Φ), if i) $\mathcal{AS} \not\vdash_X \bot$ and ii) there is no final configuration Φ' which is locally-predicate-better than Φ.

IHCS aims at computing best configurations incrementally: given an hierarchy $\mathcal{H}$ with a known best configuration $\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \emptyset \rangle$, if a new constraint c is inserted into $\mathcal{H}$, then starting from the promising configuration $\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \{c\} \rangle$ several transitions will be performed until a best configuration is reached undoing and redoing as little work as possible.

The following rules define the valid transitions for configurations. If we start with a promising configuration and a solution to the hierarchy exist, transitions will always stop at the base rule with a best configuration. While the active store is consistent and the unexplored store is not empty, the forward rule keeps activating a new constraint. If a conflict is raised (the active store becomes inconsistent) the backward rule searches for an alternative promising configuration for the hierarchy, relaxing some constraints and possibly reactivating other constraints previously relaxed. More formally,

Base rule

$$\frac{\mathcal{AS} \not\vdash_X \bot}{\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \emptyset \rangle}$$

Forward rule

$$\frac{\mathcal{AS} \not\vdash_X \bot}{\langle \mathcal{AS} \bullet \mathcal{RS} \bullet c.\mathcal{US} \rangle \rightarrow \langle \mathcal{AS}.c \bullet \mathcal{RS} \bullet \mathcal{US} \rangle}$$

Backward rule

$$\frac{\mathcal{AS} \vdash_X \bot \quad \mathcal{Relax} \subseteq (\mathcal{AS} \cup \mathcal{US}) \quad \mathcal{Activate} \subseteq \mathcal{RS} \quad \mathcal{Reset} \subseteq (\mathcal{AS} \setminus \mathcal{Relax}) \quad \mathrm{PC}(\Phi)}{\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \mathcal{US} \rangle \rightarrow \Phi}$$

$$\text{where } \Phi = \langle \mathcal{AS}' \bullet \mathcal{RS}' \bullet \mathcal{US}' \rangle \quad \begin{cases} \mathcal{AS}' = \mathcal{AS} \setminus (\mathcal{Relax} \cup \mathcal{Reset}) \\ \mathcal{RS}' = (\mathcal{RS} \setminus \mathcal{Activate}) \cup \mathcal{Relax} \\ \mathcal{US}' = (\mathcal{US} \setminus \mathcal{Relax}) \cup \mathcal{Reset} \cup \mathcal{Activate} \end{cases}$$

The main idea of the backward rule is to find constraints pertinent to the conflict that should be relaxed (the $\mathcal{Relax}$ set). Since the relaxation of these constraints may also resolve previous conflicts, constraints previously relaxed may now be re-activated (the $\mathcal{Activate}$ set). Constraints affected by the relaxed ones must be reset (temporary removed from the active store) in order to re-achieve maximum consistency (the $\mathcal{Reset}$ set). The configuration obtained $\Phi = \langle \mathcal{AS}' \bullet \mathcal{RS}' \bullet \mathcal{US}' \rangle$ must be a promising configuration, since if no other conflict is found, future transitions performed by the forward rule will lead to the final configuration $\langle \mathcal{AS}' \cup \mathcal{US}' \bullet \mathcal{RS}' \bullet \emptyset \rangle$ which will then be a best configuration.

3 Implementation of basic IHCS

IHCS is divided in two phases: a forward phase performing forward transitions where constraints are activated using an incremental arc-consistency algorithm and a backward phase corresponding to the backward rule that is evoked to solve any conflict raised during the forward phase.

3.1 The Forward Algorithm

Since methods to verify strong K-consistency are exponential for $K > 2$, [5], other weaker consistency conditions, such as Arc-consistency ($K = 2$) [6], are usually better suited for real implementations.

The forward algorithm is an adaptation of an arc consistency algorithm based on constraint propagation, generalized for the case of constraints with an arbitrary number of variables. In our implementation we adapted AC5 [10], but since arc consistency algorithms are not the main issue of this article, a simplified algorithm is described to keep this presentation clear.

The forward rule is implemented with function Forward. A counter AO is increased any time a new constraint c is inserted in the active store, to update the *activation order* of that constraint (AO_c). This order will be needed in the backward phase, as will be seen later.

A set of trail stacks are also kept to undo work in the backward phase when constraints are deactivated. For each constraint c a trail stack $\mathcal{T}_c$ is kept to record any transformation made on data structures caused by

the activation of c. The set of all trail stacks may be seen as a single partitioned trail stack. This partitioning allows to save work that is not related with the conflict raised, as it will be seen in the backward phase.

```
function Forward()
  while US = c_j.US' do
    US  ← US'
    AS  ← AS ∪ {c_j}
    AO  ← AO + 1
    AO_{c_j} ← AO
    Enqueue(c_j, Q)          % Q initially empty
    while Dequeue(Q, c_k) do
      if not Revise(c_k, T_{c_j}, Q) then
        if not Backward(c_k) then return false
  return true
```

Function Revise$(c, \mathcal{T}, Q)$ performs the removal of inconsistent values from domains of c variables and updates information about dependency between constraints (see below). All these transformations are stacked in trail $\mathcal{T}$ and all active constraints over affected variables are enqueued in Q (the propagation queue). If there are no values to satisfy c then Revise$(c, \mathcal{T}, Q)$ returns "false", otherwise "true".

When the revision of some constraint c fails, the backward algorithm will examine this dependency information to find out the pertinent causes of the failure and what will be affected by the relaxation of some constraints. The domain of variable v is denoted by D_v and for each constraint c, the set of its variables is designated by $\mathcal{V}_c$ ($\mathcal{V}_c \subseteq \mathcal{V}$).

Definition 7 (Constrainer) c ($c \in \mathcal{AS}$) is a *constrainer* of v ($v \in \mathcal{V}_c$) if it actually caused the reduction of D_v, i.e., values where removed from D_v during some revision of c.

Definition 8 (Immediate Dependent\Supporter) c_k is an *immediate dependent* of c_j (conversely c_j is an *immediate supporter* of c_k), written $c_j \hookrightarrow c_k$, iff $\exists v \in \mathcal{V}_{c_k}$ s.t. c_j is a constrainer of v.

Definition 9 (Immediate Related) c_j is *immediate related* to c_k , written $c_j \leftrightharpoons c_k$. iff $c_j \hookrightarrow c_k$ or $c_k \hookrightarrow c_j$.

A special *dependency graph* ($\mathcal{DG}$) is used to record dependencies between constraints. The implementation of $\mathcal{DG}$ and its proprieties are explained in [7]. By analyzing $\mathcal{DG}$ it is possible to compute $Supporters_c$, the set of all supporters of c (transitive closure of $\hookrightarrow$) and $Related_c$, the set of all constraints related to c (transitive closure of $\leftrightharpoons$).

The dependency relation is based on *local propagation* of constraints in the following way: whenever a constraint c_j ($c_j \in \mathcal{AS}$) makes a restriction on some $v \in \mathcal{V}_{c_j}$, any other constraint c_k ($c_k \in \mathcal{AS}$) such that $v \in \mathcal{V}_{c_k}$ will be reactivated and possibly cause the reactivation of further constraints, even if they do not share any variable with c_j. The restrictions performed by c_j may consequently affect all those constraints and for this reason they all become dependent of c_j.

3.2 The Backward Algorithm

During the backward phase, the following requisites should be attained: a) only constraints pertinent to the conflict should change status (relaxed or reactivated), to avoid un-useful search; b) a potentially best configuration must be re-achieved, to obtain a sound behavior; c) no promising configuration should be repeated, to avoid loops; d) no promising configuration should be skipped, for completeness of the algorithm; e) global consistency of the new active store must be re-achieved, undoing as little work as possible.

The relaxed store $\mathcal{RS}$ is implicitly maintained: for each active and unexplored constraint c, an *opponent constraints* set is kept ($\mathcal{OC}_c$) with all the relaxed constraints with whom c had previous conflicts and $\mathcal{RS} = \bigcup_{c \in \mathcal{AS} \cup \mathcal{US}} \mathcal{OC}_c$. This representation allows an easy access to candidates for re-activation, should c be relaxed. The hierarchical level of c is expressed by $level_c$.

function Backward(**in** c_j)

$\mathcal{AS}_{\text{conf}} \leftarrow \{c_k \in \mathcal{AS} \mid level_{c_k} > 0 \text{ and } c_k \in Supporters_{c_j}\}$ % Step1)

if $\mathcal{AS}_{\text{conf}} = \emptyset$ **then return** false % Conflict

else

$\mathcal{US}_{\text{conf}} \leftarrow \{c \in \mathcal{US} \mid level_c > 0\}$ % Configuration

$\mathcal{RS}_{\text{conf}} \leftarrow \bigcup_{c \in \mathcal{AS}_{\text{conf}} \cup \mathcal{US}_{\text{conf}}} \mathcal{OC}_c$ %

ActivateRelaxSets($\langle \mathcal{AS}_{\text{conf}} \cup \mathcal{US}_{\text{conf}} \bullet \mathcal{RS}_{\text{conf}} \rangle, Activate, \mathcal{R}elax$) % Step2) Activate & Relax

$\mathcal{R}eset \leftarrow \{c_k \in \mathcal{AS} \mid \exists c_j \in \mathcal{R}elax,\ AO_{c_k} > AO_{c_j} \text{ and } c_k \in Relateds_{c_j}\}$ % Step3) Reset Set

untrail($\mathcal{T}_c$) ($\forall c \in \mathcal{R}eset \cup \mathcal{R}elax$) % Step4) Untrailing

$\mathcal{AS} \leftarrow \mathcal{AS} \setminus (\mathcal{R}eset \cup \mathcal{R}elax)$ % Step5)

$\mathcal{OC}_c \leftarrow \mathcal{OC}_c \setminus Activate$ ($\forall c \in \mathcal{C}$) % New

$\mathcal{OC}_c \leftarrow \mathcal{OC}_c \cup \mathcal{R}elax$ ($\forall c \in (\mathcal{AS}_{\text{conf}} \cup \mathcal{RS}_{\text{conf}} \cup \mathcal{US}_{\text{conf}})$) % Configuration

$\mathcal{US} \leftarrow (\mathcal{US} \setminus \mathcal{R}elax) \cup \mathcal{R}eset \cup Activate$ %

return true

Conflict configuration. Step 1 of the backward algorithm computes the *conflict configuration* $\Phi_{\text{conf}} = \langle \mathcal{AS}_{\text{conf}} \bullet \mathcal{RS}_{\text{conf}} \bullet \mathcal{US}_{\text{conf}} \rangle$, which includes only those constraints pertinent to the conflict ($\mathcal{AS}_{\text{conf}} \subseteq \mathcal{AS}$, $\mathcal{RS}_{\text{conf}} \subseteq \mathcal{RS}$ and $\mathcal{US}_{\text{conf}} \subseteq \mathcal{US}$). Φ_{conf} is thus the only portion of the whole configuration that should be changed to solve the conflict. $\mathcal{AS}_{\text{conf}}$ and $\mathcal{US}_{\text{conf}}$ are the candidates for relaxation and $\mathcal{RS}_{conf}$ the candidates for re-activation. Note that although $\mathcal{US}_{\text{conf}}$ does not contain any active constraint, some of them may have to be relaxed (in this case, no longer activated) to ensure that all promising configurations will be tried.

The possible causes of the conflict are all the supporters of c_j (the failing constraint). Those supporters represent the constraints that directly or indirectly restricted the domains of the variables of c_j, so that no consistent values remained to satisfy c_j. Since required constraints may not be relaxed, $\mathcal{AS}_{\text{conf}}$ will only include the non-required supporters of the failing constraint. If $\mathcal{AS}_{\text{conf}}$ is empty then there is no possible solution to the conflict and the constraint hierarchy is not satisfiable.

The backward rule is the only rule that inserts constraints in the unexplored store. If the current conflict is not the first to occur, then during the resolution of the previous conflict the backward rule generated a promising configuration with unexplored constraints. After some transitions, that configuration proved to be not convertible into a best configuration since it lead to the current conflict. The current conflict is thus related to the previous one and $\mathcal{US}_{\text{conf}}$ is formed by all non required constraints left unexplored.

Constraints relaxed in previous conflicts should be reconsidered for re-activation, if some constraints involved in those conflicts are now relaxed. There is a chance that the new relaxations will also solve those early conflicts hence allowing previously relaxed constraints to be active now. $\mathcal{RS}_{\text{conf}}$ is the set of candidates for re-activation which are the opponents in previous conflicts of any candidate for relaxation.

Activate and Relax Sets. By analyzing the conflict configuration, this step determines which non relaxed constraints should be relaxed (the $\mathcal{R}elax$ set), and which relaxed constraints should be activated (the *Activate* set) in order to obtain the next promising configuration. Since unexplored constraints are candidates for activation, one can consider only two states in which a constraint can be: relaxed or non relaxed. Therefore, step 2 of the backward algorithm only manipulates a simplified form of configurations with only two stores, $\langle \mathcal{NS} \bullet \mathcal{RS} \rangle$, where the first store contains the non relaxed constraints (active or unexplored) and the second one the relaxed constraints.

As mentioned before, given a constraint store S, S_i designates the subset of S containing all constraints of level i. $S_{<i}$ and $S_{>i}$ are the subsets of S containing all constraints of levels lower and higher then i respectively. Given a configuration $\Phi = \langle \mathcal{NS} \bullet \mathcal{RS} \rangle$, its levels i is denoted by $\Phi_i = \langle \mathcal{NS}_i \bullet \mathcal{RS}_i \rangle$.

The $\mathcal{LPB}$ ordering is not a total ordering since some configurations are not comparable. A total ordering is nevertheless necessary to enssure a sound and complete search for solutions without generating the same configuration more than once. Definition 10 is a refinement of Definition 4, adapted for simplified configurations and taking into account *introduction orders*. In case of ambiguity between configurations not $\mathcal{LPB}$ comparable, i.e. having exactly the same number of non relaxed constraints in each level, the *introduction orders* of constraints are used in Condition 2 to determine the "best" configuration.

Definition 10 (Extended $\mathcal{LPB}$) $(\mathcal{NS} \bullet \mathcal{RS})$ is locally-predicate-better then $(\mathcal{NS}' \bullet \mathcal{RS}')$ if exist some level k such that:

1. $\forall i < k$, $\#\mathcal{NS}_i = \#\mathcal{NS}'_i$ and $\#\mathcal{NS}_k > \#\mathcal{NS}'_k$

or

2. $\forall i, \#\mathcal{NS}_i = \#\mathcal{NS}'_i$ and

   a. $\mathcal{NS}_{<k} = \mathcal{NS}'_{<k}$;

   b. $\exists c_i \in \mathcal{NS}_k$, $\exists c_j \in \mathcal{NS}'_k$ s.t. $\{c_l \in \mathcal{NS}_k \mid l < i\} = \{c_l \in \mathcal{NS}'_k \mid l < j\}$ and $i < j$.

Definition 11 (Restarted Level) A configuration level $\Phi_i = (\mathcal{NS}_i \bullet \mathcal{RS}_i)$ is *restarted*, if $\forall c_j \in \mathcal{RS}_i, \forall c_k \in \mathcal{NS}_i, j > k$.

Informally a restarted level is the first permutation of constraints of that level with the same number of relaxed constraints which is generated to access its satisfiability.

Example 1 Given $\Phi = (\{c_1@1, c_2@1, c_3@2, c_6@3\} \bullet \{c_4@2, c_5@3, c_7@4, c_8@4\})$, levels $\Phi_1 = (\{c_1, c_2\} \bullet \emptyset)$, $\Phi_2 = (\{c_3\} \bullet \{c_4\})$ and $\Phi_4 = (\emptyset \bullet \{c_7, c_8\})$ are all restarted.

Definition 12 (Exhausted Level) A configuration level $\Phi_i = (\mathcal{NS}_i \bullet \mathcal{RS}_i)$ is *exhausted*, if $\forall c_j \in \mathcal{RS}_i$, $\forall c_k \in \mathcal{NS}_i, j < k$.

Informally, all permutation of constraints of an exhausted level with the same number of relaxed constraints have been (potentially) generated and tested before.

Example 2 Given configuration Φ of Example 12, levels $\Phi_1 = (\{c_1, c_2\} \bullet \emptyset)$, $\Phi_3 = (\{c_6\} \bullet \{c_5\})$ and $\Phi_4 = (\emptyset \bullet \{c_7, c_8\})$ are all exhausted.

Procedure ActivateRelaxSets, used in Step 2 of the backward algorithm, computes the *Activate* and *Relax* sets in order to obtain the successor of $(\mathcal{AS}_{conf} \cup \mathcal{US}_{conf} \bullet \mathcal{RS}_{conf})$ according to the extended $\mathcal{LPB}$ ordering. This procedure is fully defined in [7]. Here we simply present examples of its behavior in three illustrative situations.

Example 3 The successor of $\Phi = (\{c_1@1, c_2@2, c_3@3\} \bullet \{c_4@1, c_5@2, c_6@3\})$ (in the extended $\mathcal{LPB}$ ordering) is $(\{c_1@1, c_2@2, c_6@3\} \bullet \{c_3@3, c_4@1, c_5@2\})$. This is the simplest situation where the highest level (the third) is not exhausted and thus it is the only level to be changed to the next permutation with the same number of relaxed constraints. From input Φ, procedure ActivateRelaxSets produces $\mathcal{Activate} = \{c_6\}$ and $\mathcal{Relax} = \{c_3\}$.

Example 4 The successor of $\Phi = (\{c_1@1, c_2@2, c_6@3\} \bullet \{c_3@3, c_4@1, c_5@2\})$ is $(\{c_1@1, c_3@3, c_5@2\} \bullet \{c_2@2, c_4@1, c_6@3\})$. Here the number of relaxed constraints in each level is also kept but the highest non exhausted level (the second) is an intermediate level. Level 2 is thus changed to the next permutation with the same number of relaxed constraints and level 3 is *restarted*. Procedure ActivateRelaxSets produces $\mathcal{Activate} = \{c_3, c_5\}$ and $\mathcal{Relax} = \{c_2, c_6\}$ from input Φ.

Example 5 The successor of $\Phi = (\{c_2@2, c_4@1, c_5@2\} \bullet \{c_1@1, c_3@3, c_6@3\})$ is $(\{c_1@1, c_2@2, c_3@3, c_6@3\} \bullet \{c_4@2, c_5@3\})$. This is an extreme example where all levels are exhausted. Level 2 is the highest level with a relaxable constraint and thus it is restarted with one extra relaxed constraint. Level 1 is restarted with the same number of relaxed constraints and all relaxed constraints of level 3 are activated. A worst configuration is thus obtained. From input Φ, procedure ActivateRelaxSets produces $\mathcal{Activate} = \{c_1, c_3, c_6\}$ and $\mathcal{Relax} = \{c_4, c_5\}$.

Reset Set. The *Reset* set contains all constraints that will have to be reset, i.e. moved from the active store to the unexplored store (for subsequent activation "from scratch"), to re-achieve global consistency. If c_j is one of the active constraints to be relaxed and c_k is a constraint activated after c_j ($AO_{c_k} > AO_{c_j}$), then c_k needs to be reset if it is related to c_j. Either directly or indirectly by constraint propagation, c_j has caused the removal of some values from the domains of c_k variables, that otherwise c_k itself would be charged to remove. Therefore c_k will have to be reset to perform such removal. Alternatively, c_j made more removals after the activation of c_k. Therefore c_k will have to be reset, to be used in a context without such removals have taken place.

Example 6 Given variables X and Y with initial domains $D_X = D_Y = 1..10$, consider the following constraints: $c_1 \equiv X + Y = 15@1$; $c_2 \equiv 3 \cdot X - Y < 5@1$; $c_3 \equiv X > Y + 1@2$; $c_4 \equiv X < 7@2$. The incremental insertion of each constraint is given by the following transitions:

Action	Configuration	D_X	D_Y	Relaxed @1	Relaxed @2	Rule
insert c_1	$\langle \emptyset \bullet \emptyset \bullet \{c_1\} \rangle$	1..10	1..10	0	0	forward
	$\langle \{c_1\} \bullet \emptyset \bullet \emptyset \rangle$	5..10	5..10	0	0	base
insert c_2	$\langle \{c_1\} \bullet \emptyset \bullet \{c_2\} \rangle$	5..10	5..10	0	0	forward
	$\langle \{c_1, c_2\} \bullet \emptyset \bullet \emptyset \rangle$	$\emptyset$	$\emptyset$	0	0	backward
	$\langle \{c_1\} \bullet \{c_2\} \bullet \emptyset \rangle$	5..10	5..10	1	0	base
insert c_3	$\langle \{c_1\} \bullet \{c_2\} \bullet \{c_3\} \rangle$	5..10	5..10	1	0	forward
	$\langle \{c_1, c_3\} \bullet \{c_2\} \bullet \emptyset \rangle$	7..10	5..8	1	0	base
insert c_4	$\langle \{c_1, c_3\} \bullet \{c_2\} \bullet \{c_4\} \rangle$	7..10	5..8	1	0	forward
	$\langle \{c_1, c_3, c_4\} \bullet \{c_2\} \bullet \emptyset \rangle$	$\emptyset$	$\emptyset$	1	0	backward
	$\langle \{c_1, c_3\} \bullet \{c_2, c_4\} \bullet \emptyset \rangle$	7..10	5..8	1	1	base

4 Extensions to Basic IHCS

In addition to the basic transition rules and algorithms, a number of extensions are incorporated in IHCS, allowing it to search for alternative solutions, to incrementally remove constraints and to cope with disjunctions of constraints.

4.1 Obtaining alternative solutions

Several best configurations of an hierarchy may exist since $\mathcal{LPB}$ ordering is not a total ordering. Given $\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \emptyset \rangle$, the current best configuration, IHCS is able to find the next promising configuration, with a slightly modified backward rule (the *alternative* rule).

Alternative rule

$$\frac{\mathcal{AS} \not\vdash_X \bot \quad \mathcal{Relax} \subseteq \mathcal{AS} \quad \mathcal{Activate} \subseteq \mathcal{RS} \quad \mathcal{Reset} \subseteq (\mathcal{AS} \setminus \mathcal{Relax}) \quad \mathrm{PC}(\Phi)}{\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \emptyset \rangle \rightarrow \Phi}$$

where $\Phi = \langle \mathcal{AS} \setminus (\mathcal{Relax} \cup \mathcal{Reset}) \bullet \mathcal{RS} \setminus \mathcal{Activate} \cup \mathcal{Relax} \bullet \mathcal{Reset} \cup \mathcal{Activate} \rangle$

This rule is implemented using the backward algorithm with slight modifications in Steps 1 and 2 in order to find an alternative to a best configuration rather than to a conflicting one. In Step 1 all active non required constraints must be considered instead of only the supporters of a failing constraint as in the normal backward rule. In Step 2 the search for an alternative fails if an extra relaxed constraint is required ("if" branch of procedure ActivateRelaxSets) since the new configuration would be worse than the known current best configuration.

The promising configuration Φ determined by the alternative rule will be input to the normal set of rules (the base, forward and backward rules) to find a best configuration. If, at any, point an intermediate configuration is worse than the previous one, the search for an alternative best solution fails.

Example 7 An alternative to the solution found in Example 6 is computed by the following transitions:

Configuration	D_X	D_Y	Relaxed @1	Relaxed @2	Rule
$\langle\{c_1, c_3\} \bullet \{c_2, c_4\} \bullet \emptyset\rangle$	7..10	5..8	1	1	alternative
$\langle\{c_1\} \bullet \{c_2, c_3\} \bullet \{c_4\}\rangle$	5..10	5..10	1	1	forward
$\langle\{c_1, c_4\} \bullet \{c_2, c_3\} \bullet \emptyset\rangle$	5..6	9..10	1	1	base

4.2 Incremental Removal of Constraints

Given a best configuration $\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \emptyset\rangle$ to a hierarchy $\mathcal{H}$, the removal of a constraint c from $\mathcal{H}$ is straightforward if c is a relaxed constraint, and a best configuration is immediatly obtained by the following rule:

Relaxed Removal rule

$$\frac{\mathcal{AS} \not\vdash_X \bot \qquad c \in \mathcal{RS}}{\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \emptyset\rangle \rightarrow \langle \mathcal{AS} \bullet \mathcal{RS} \setminus \{c\} \bullet \emptyset\rangle}$$

If c is active, a slightly modified backward rule is necessary to obtain a promising configuration:

Active Removal rule

$$\frac{\mathcal{AS} \not\vdash_X \bot \quad c \in \mathcal{AS} \quad \mathit{Activate} \subseteq \mathcal{RS} \quad \mathit{Reset} \subseteq (\mathcal{AS} \setminus \{c\}) \quad \mathrm{PC}(\Phi)}{\langle \mathcal{AS} \bullet \mathcal{RS} \bullet \emptyset\rangle \rightarrow \Phi}$$

where $\Phi = \langle \mathcal{AS} \setminus (\{c\} \cup \mathit{Reset}) \bullet \mathcal{RS} \setminus \mathit{Activate} \bullet \mathit{Reset} \cup \mathit{Activate}\rangle$

This rule is implemented following Steps 3 to 5 of the backward algorithm, provided that $\mathcal{Relax} = \{c\}$ and $\mathcal{Activate} = \mathcal{OC}_c$ and that c is not included in the relaxed store obtained. Feeding Φ to the normal set of transition rules will lead to a best configuration.

Example 8 Removing the active constraint c_4 from the configuration obtained in Example 7 produces the following transitions:

Configuration	D_X	D_Y	Relaxed @1	Relaxed @2	Rule
$\langle\{c_1, c_4\} \bullet \{c_2, c_3\} \bullet \emptyset\rangle$	5..6	9..10	1	1	active removal
$\langle\{c_1\} \bullet \{c_2\} \bullet \{c_3\}\rangle$	5..10	5..10	1	0	forward
$\langle\{c_1, c_3\} \bullet \{c_2\} \bullet \emptyset\rangle$	7..10	5..8	1	0	base

4.3 Disjunctive Constraints and Inter-Hierarchies

In logic programming the alternatives to solve a goal are usually specified as different rules for the same literal. This fact raises some problems, since different choices of rules in the logic program may produce solutions arriving from different constraint hierarchies, sometimes producing non-intuitive solutions. In [11] the HCLP scheme is extended with some non-monotonic properties of comparators to cope with inter-hierarchy comparisons.

This problem was dealt with in IHCS by extending it with disjunctive constraints of the form $c = c^1 \bigvee c^2 \bigvee \cdots \bigvee c^n$, where c^i is a normal constraint representing the ith alternative of c. c can only be relaxed if $\mathit{level}_c > 0$ and all alternatives have already failed.

This extension, which is formalized in [7], enables the specification of more complex constraint hierarchies and we take advantage of the dependency graph to backtrack intelligently to alternative choices.

Disjunctions however complicates the overall IHCS algorithm, as non exhausted disjunctions must be integrated in conflict configurations. Since we want to minimize the number of constraints to be relaxed, it is preferable whenever possible to try an alternative choice rather than relaxing extra constraints. As in intelligent backtracking methods, an *alternative set* is associated to each disjunction – cf. the *Alt* sets of [4] – and disjunctions to be re-inserted are restarted from the first alternative – cf. the selective reset of [3].

The use of disjunctive constraints is very useful for the final generation of solutions. After the pruning due to all constraints being treated, some variables may still have several possible values in their domains. If the domain of a variable v is $\{w_1, \ldots, w_n\}$ then adding a constraint $v = w_1 \bigvee \cdots \bigvee v = w_n$ will assure that a single value will be assigned to v within a best solution. We used such constraint as the basic definition for a built-in value generator - predicate *indomain(v)*.

5 Applications

We integrated IHCS with prolog to create a HCLP($\mathcal{FD}, \mathcal{LPB}$) language, using pre-processing methods. At present we are employing YAP prolog running on a NeXT Station 68040.

In this section we describe two problems with our HCLP language, namely a set-covering problem and a time-tabling problem, to illustrate the applicability and declarativity of hierarchical constraints and the efficiency of our incremental approach to solve them.

In the set-covering problem, the goal is to minimize the number of services required to cover a set of needs (the problem variables designated by $X_1, \ldots, X_m$). Each variable ranges over the set of services that cover that need. The approach that we took to solve this problem is depicted in the following HCLP program:

cover($[X_1, \ldots, X_m]$) :-
$X_1 = X_2 \bigvee X_1 = X_3 \bigvee \cdots \bigvee X_1 = X_m$ @ 1,
$X_2 = X_3 \bigvee X_2 = X_4 \bigvee \cdots \bigvee X_2 = X_m$ @ 1,
$\vdots$
$X_{m-1} = X_m$ @ 1,
labeling($[X_1, \ldots, X_m]$).

For m needs, predicate *cover/1* states $m - 1$ disjunctive constraints of level 1. This set of constraints will try to assure that the service assigned to variable X_i will also be assigned to at least some X_j, $j > i$. Predicate *labeling/1* simply uses the built-in predicate *indomain* to generate values for each variables. A best solution (one that relaxes the minimum of constraints as possible) will correspond to the minimization of services. Table 1 presents results obtained using several real life instances, taken from a Portuguese Bus company. The time presented concerns the first (best) solution found and column *Min* reports the minimum number of services required to cover the needs.

Table 1: Results for the set-covering problem

Needs	Services	Time	Min
13	43	0.33s	6
24	293	3.98s	7
38	67	3.57s	11

The time-tabling problem is taken from the experience in the Computer Science Department of UNL, but it is simplified so that no spatial constraints are considered (it is assumed that there are enough rooms) and each subject is already assigned to a teacher (c.f. [8] for a full description). For this problem we used a multi-level hierarchy to model preferences of different strength regarding the layout of blocks of subjects in a time-table. Table 2 presents results for the generation of time-tables for three semesters. The first line reports the results obtained by specifying only required constraints (teachers availability, blocks for the same subject at different days, non overlapping of classes for the same semester or given by the same teacher). Each of the other lines shows the effect of adding an extra hierarchical level.

Constraints in each level are designated to: level 1) avoid consecutive blocks of a subject at consecutive days; level 2) avoid consecutive blocks of a subject to be apart by more than two days; level 3) disallow any block, from a subject with only two blocks per week, to take place on mondays; 4) place blocks of the same subject at the same hour. The *Relaxed Constraints* columns report the number of preferred constraints

Table 2: Results for the time-tabling problem

Max. level	Number of constraints	Time	Relaxed Constraints @1	@2	@3	@4
0	356	1.80s	(16)	(1)	(7)	(15)
1	+21 = 377	1.86s	2	(4)	(7)	(11)
2	+21 = 398	1.98s	2	1	(5)	(10)
3	+11 = 409	1.98s	2	1	1	(10)
4	+21 = 430	2.33s	2	1	1	0

relaxed in each level (in fact, values inside round brackets do not represent relaxed constraints, since that level was not being used, but rather the number of those that are not satisfied by the solution).

The introduction of the preferred constraints of each level, significantly increases the quality of the solution. The last one satisfies 95% of the preferences against only 47% satisfied by the first one with a mere slowdown penalty of 32%.

6 Conclusion

This paper reports a first formalization of IHCS as a set of transition rules over hierarchy configurations. We conjecture that the forward and backward algorithms presented are a) sound (only locally predicate better solutionsare obtained), b) complete (all such solutions are computed), and c) non redundant (no repeated solutions). These properties are yet to be formally proven, and this task is in our plans for future work.

The experimental results shown in the examples, are quite promising with respect to IHCS performance. Yet, the algorithm complexity (both in time and memory requirements) is yet to be fully assessed. This is likely to be related with the study for a potential replacement of the present criterion (locally-predicate-better), which aims at finding some kind of optimal solutions, by a less demanding satisfiability criterion (e.g. a solution is acceptable if a certain threshold of preferences is met).

References

[1] P. Barahona and R. Ribeiro. Building an Expert Decision Support System: The Integration of AI and OR methods. In *Knowledge, Data and Computer-Assisted Decisions*. Springer-Verlag, Berlin Heidelberg, 1990.

[2] A. Borning, M. Maher, A. Martingale, and M. Wilson. Constraints hierarchies and logic programming. In *Proceedings of 6th ICLP*, Lisbon, 1989. The MIT press.

[3] C. Codognet and P. Codognet. Non-deterministic Stream AND-Parallelism based on Intelligent Backtracking. In *Proceedings of 6th ICLP*, Lisbon, 1989. The MIT press.

[4] C. Codognet, P. Codognet, and G. Filé. Yet Another Intelligent Backtracking Method. In *Proceedings of 5th ICLP/SLP*, Seattle, 1988.

[5] Vipin Kumar. Algorithms for Constraint-Satisfaction-Problems: A Survey. *AI Magazine*, Spring 1992.

[6] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[7] F. Menezes and P. Barahona. Report on IHCS. Research report, Universidade Nova de Lisboa, 1993.

[8] F. Menezes, P. Barahona, and P. Codognet. An Incremental Hierarchical Constraint Solver Applied to a Time-tabling Problem. In *Proceedings of Avignon 93*, 1993. Forthcomming.

[9] Luis Moniz Pereira and M. Bruynooghe. Deduction Revision by Intelligent Backtracking. In *Implementations of Prolog*. J.A. Campbell, 1984.

[10] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A Generic Arc Consistency Algorithm and its Specializations. Technical Report RR 91-22, K.U. Leuven, F.S.A., December 1991.

[11] M. Wilson and A. Borning. Extending Hierarchical Constraint Logic Programming: Nonmonotonocity and Inter-Hierarchy Comparison. In *Proceedings of the North American Conference 1989*, 1989.

Constraining the Structure and Style of Object-Oriented Programs

Scott Meyers
Dept. of Computer Science
Brown University, Box 1910
Providence, RI 02912
sdm@cs.brown.edu

Carolyn K. Duby
Cadre Technologies, Inc.
222 Richmond Street
Providence, RI 02903
ckd@cadre.com

Steven P. Reiss
Dept. of Computer Science
Brown University, Box 1910
Providence, RI 02912
spr@cs.brown.edu

Abstract

Object-oriented languages fail to provide software developers with a way to say many of the things about their systems that they need to be able to say. To address this need, we have designed and implemented a language for use with C++ that allows software developers to express a wide variety of constraints on the designs and implementations of the systems they build. Our language is specifically designed for use with C++, but the issues it addresses are applicable to other object-oriented languages, and the fundamental software architecture used to implement our system could be applied without modification to similar constraint languages for other object-oriented programming languages.

1 Introduction

C++ is an expressive language, but it does not allow software developers to say many of the things about their systems that they need to be able to say. In particular, C++ offers no way to express many important constraints on a system's design, implementation, and stylistic conventions. Consider the following sample constraints, none of which can be expressed in C++:

- *The member function M in class C must be redefined in all classes derived from C.* This is an example of a **design constraint**, because the constraint is specific to a particular class, C, and a particular function in that class, M. This kind of constraint is common in general-purpose class libraries. For example, the NIH class library [6] contains many functions which must always be redefined if the library is to function correctly.

- *If a class declares a pointer member, it must also declare an assignment operator and a copy constructor.* This is an example of design-independent **implementation constraint**. Failure to adhere to this constraint almost always leads to incorrect program behavior [16].

- *All class names must begin with an upper case letter.* This is an example of one of the most common kinds of **stylistic constraint**. Most software development teams adopt some set of naming conventions that developers are required to follow.

Constraints such as these exist (usually only implicitly) in virtually every system implemented in C++, but different systems require very different sets of constraints. That fact makes it untenable for C++ compilers to search for constraint violations. Our approach to this problem is the development of a new language, CCEL ("Cecil") — the C++ Constraint Expression Language — a language for use with C++ that allows

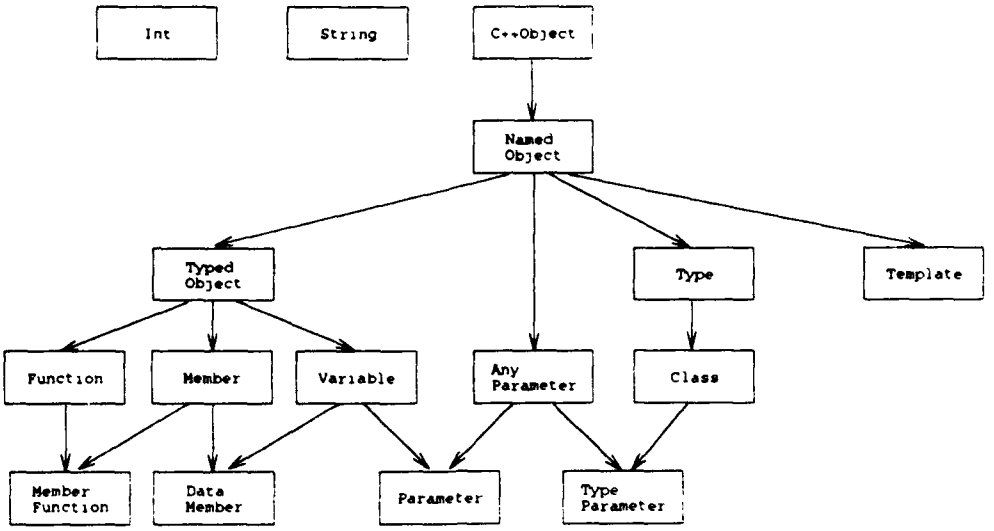


Figure 1: CCEL Class Hierarchy

software developers to specify a wide variety of constraints and to automatically detect violations of those constraints.

We took as our original inspiration the lint tool for C programmers, but we quickly discovered that the kinds of errors C programmers need to detect are qualitatively different from the errors that C++ programmers need to detect [17]. lint concentrates on type mismatches and data-flow anomalies, but the stronger typing of C++ obviates the need for lint's type-checking, and data flow analysis is typically unrelated to the high-level perspective encouraged by the modular constructs of C++. Instead, C++ programmers are concerned with concepts such as the structure of a design-specific inheritance hierarchy, but lint offers no provision for design-specific error detection. The most fundamental difference, then, between lint and CCEL is that CCEL was designed from the outset to allow for the addition of *programmer-defined* constraints.

An earlier working paper describing CCEL has already been published [4], but since the time of that publication we have made significant improvements to both the language and the software architecture built around it. Such improvements include a simpler, more uniform syntax; a set of predefined types that is more expressive and is easier to understand and use; and an implementation architecture that is more portable, more robust, and has a more convenient user interface.

2 Design Considerations, Syntax, and Semantics

Our primary requirements in designing CCEL were these:

- It must offer sufficient expressive power to allow programmers to specify a wide variety of constraints, including constraints on both concrete syntax (stylistic constraints) and semantics (design and implementation constraints).
- It must be relatively easy to learn and use. In particular, the syntax and semantics of the language should mesh well with the syntax and semantics of C++. We chose as our point of departure C++ itself and the well-known assert macro.

Like the programs C++ is used to build, CCEL has an object-oriented basis. CCEL classes represent the concepts of C++. The CCEL classes are arranged in a multiple inheritance "isa" hierarchy (see Figure 1), and each class supports a particular set of member functions (see Table 1).

Syntactically, CCEL constraints resemble expressions in the predicate calculus, allowing programmers to make assertions involving existentially or universally quantified CCEL variables. In general, constraint

CCEL Class	Member Function
Int	Int operator == (Int)
	Int operator < (Int)
	Int operator ! ()
	Int operator && (Int)
	Int operator || (Int)
	Int operator != (Int)
	Int operator > (Int)
	Int operator <= (Int)
	Int operator >= (Int)
String	Int operator == (String)
	Int operator < (String)
	Int matches(String)
	String operator + (String)
	Int operator != (String)
	Int operator > (String)
	Int operator <= (String)
	Int operator >= (String)
C++Object	String file()
	Int begin_line()
	Int end_line()
NamedObject	String name()
TypedObject	Type type()
	Int num_indirections()
	Int is_reference()
	Int is_static()
	Int is_volatile()
	Int is_const()
	Int is_array()
	Int is_long()
	Int is_short()
	Int is_signed()
	Int is_unsigned()
	Int is_pointer()

CCEL Class	Member Function
Type	Int has_name(String)
	Type basic_type()
	Int operator == (Type)
	Int is_convertiable_to(Type)
	Int is_enum()
	Int is_class()
	Int is_struct()
	Int is_union()
	Int is_friend(Class)
	Int is_child(Class)
	Int is_descendant(Class)
	Int is_virtual_descendant(Class)
	Int is_public_descendant(Class)
	Int operator != (Type)
Template	Int is_class_template()
	Int is_function_template()
Function	Int is_global()
	Int num_params()
	Int is_inline()
	Int is_friend(Class)
Member	Int is_private()
	Int is_protected()
	Int is_public()
Variable	Int scope_is_file()
	Int scope_is_local()
AnyParameter	Int position()
Class	
MemberFunction	Int is_virtual()
	Int is_pure_virtual()
	Int overrides(MemberFunction)
DataMember	
Parameter	Int has_default_value()
TypeParameter	

Table 1: CCEL Class Member Functions

violations are reported for each combination of CCEL variable bindings that causes an assertion to evaluate to false.

There are five parts to a CCEL constraint:

1. A unique identifer that serves as the name of the constraint.
2. A set of declarations for universally quantified CCEL variables. Such variables take as their values components of C++ programs. Each CCEL variable has a type; this type is one of the CCEL classes shown in Figure 1.
3. An assertion that comprises the essence of the constraint. Assertions may use universally quantified CCEL variables and may declare and use existentially quantified CCEL variables.
4. A scope specification that determines the region of applicability of the constraint in relation to the C++ source being checked. By default, constraints are globally applicable, but they may be restricted to a single file, function, or class.
5. A message to be issued when a violation of the constraint is detected.

Of these five parts, only the constraint identifier and the assertion are required. If omitted, the set of CCEL variables is empty, the scope of applicability is global, and constraint violations are indicated by a message in a default format.

As an example of a CCEL constraint, consider Meyers' admonition [16] that every base class in C++ should declare a virtual destructor:

```
BaseClassDtor (                          This constraint is called "BaseClassDtor":
  Class B;                               For all classes B,
  Class D | D.is_descendant(B);          for all classes D such that D is a descendant of B,
  Assert(MemberFunction B::m; |          there must exist a member function m in B such that
        m.name() == "~" + B.name() &&    m's name is a tilde followed by B's name and
        m.is_virtual());                 m is a virtual function.
  );
```

Within this constraint, the variables B and D are universally quantified,[1] and m is existentially quantified. The scope of the constraint has been omitted, so it applies to all classes. The violation message has also been omitted. This constraint is in fact an important one in practice, because programs that violate it, though legal, almost always behave incorrectly [17].

The assertion inside a constraint is evaluated only if it is possible to find a binding for each of the universally quantified variables declared in the constraint. It can therefore be useful to write constraints containing assertions that always fail, the goal being to detect the ability to bind to a universally quantified variable. For example, class templates in C++ may take either type or non-type parameters, but function templates may take only type parameters, and this inconsistency (as well as other considerations) may make it advisable to avoid non-type parameters in templates of any kind. This rule can be formalized in CCEL as follows:

```
// Templates should take only type parameters:
TypeParametersOnly (
  Template t;                            For all templates t and
  Parameter t<p>;                        all non-type parameters p of t
  Assert(FALSE);                         issue a violation message.
);
```

Individual constraints are useful, but it is often convenient to group constraints together. This is especially the case with stylistic constraints, because a consistent style can typically be achieved only through adherence to a number of individual constraints. CCEL provides for this grouping capability through support for *constraint classes*, which comprise a set of individual constraints. For example, suppose there are several constraints on naming conventions. They could be grouped together into a constraint class called NamingConventions:

```
NamingConventions {
  // Every class name must begin with an upper case letter:
  CapitalizeClassNames (
        Class C; // C is a class, struct, or union
        Assert(C.name().matches("^[A-Z]"));
  );

  // Every function name must begin with a lower case letter:
  SmallFunctionNames (
        Function F;
        Assert(F.name().matches("^[a-z]"));
  );
};
```

[1] B and D are both of type Class, which technically corresponds to C++ classes, structs, and unions, i.e., any language construct that may contain member functions. Hence B and D may be bound to any of these language constructs. This use of the term "class" is consistent with that employed by the C++ language reference manual [5]. Member functions in the CCEL class Class allow programmers to distinguish between classes, structs, and unions.

Notice that the extent of constraint classes is demarcated by brackets {...}, while individual constraints use parentheses as their delimiters.

Sometimes what is a single conceptual constraint is best expressed using a set of simpler constraints bundled into a constraint class. The following example consists of a pair of constraints that detects undeclared assignment operators for classes that contain a pointer member or are derived from classes containing a pointer member. This is an important constraint in real-world C++ programs [16] and is one that cannot be specified in C++ itself:

```
PointersAndAssignment {
  // If a class contains a pointer member, it must declare an assignment operator:
  AssignmentMustBeDeclaredCond1 (
        Class C;
        DataMember C::cmv | cmv.is_pointer();
        Assert(MemberFunction C::cmf; | cmf.name() == "operator=");
  );

  // If a class inherits from a class containing a pointer member, the
  // derived class must declare an assignment operator:
  AssignmentMustBeDeclaredCond2 (
        Class B;
        Class D | D.is_descendant(B);
        DataMember B::bmv | bmv.is_pointer();
        Assert(MemberFunction D::dmf; | dmf.name() == "operator=");
  );
};
```

By default, a constraint applies to all code in the system. This is not always desirable. For example, a programmer might have one set of naming conventions for a class library and a different set of naming conventions for application-specific classes. CCEL explicitly provides for the need to restrict the applicability of constraints to subsets of the system being checked. In particular, the scope of a constraint may be restricted to any named portion of a C++ system: a file, a function, or a class. For files, scopes are specified in terms of UNIX shell wildcard expressions. For functions and classes, scopes are specified in terms of UNIX regular expressions. For example, if we wanted to limit the applicability of CapitalizeClassNames to the file file.h, we could declare a scope for the constraint. Such scope specifications precede the name of the constraint:

```
  // The name of every class declared in the file "file.h" must begin with a capital letter:
  File "file.h": CapitalizeClassNames (
        Class C;
        Assert(C.name().matches("^[A-Z]"));
);
```

Sometimes it is more convenient to specify where an otherwise global constraint does *not* apply. This can be accomplished by creating a new constraint with a restricted scope of application. The new constraint does nothing but *disable* the constraint that should not apply to the specified scope. Such disabling occurs through the Disable keyword. For example, to set things up so that CapitalizeClassNames applies to every C++ class *except* class X, we could disable CapitalizeClassNames for that class (note the use of a regular expression to specify only the class name "X"):

```
// Do not report violations of CapitalizeClassNames in class X:
Class "^X$" : DontCapitalizeInX (
  Disable CapitalizeClassNames;
);
```

Like individual constraints, constraint classes may be disabled. This is most frequently combined with a scoping specification:

```
// Ignore naming conventions for the file "importedFromC.h":
File "importedFromC.h" : NamingConventionsOff (
  Disable NamingConventions;
);
```

Individual members of a constraint class may be disabled by referring to them using the C++ scoping operator("::"):

```
// Turn off the constraint NamingConventions::CapitalizeClassNames for the file file.h only:
File "file.h" : SomeNamingConventionsOff (
  Disable NamingConventions::CapitalizeClassNames;
);
```

When a constraint violation is detected, a message to that effect is issued identifying the location of the CCEL constraint, the name of the C++ object bound to each universally quantified variable, and the location of each object so bound. Locations consist of a file name and a line number corresponding to the beginning of the source code for the object being identified. For example, consider this declaration for an array template:

```
template<class T, int size> class BoundedArray { ... };
```

This template violates the TypeParametersOnly constraint discussed earlier, so a violation message in the following format would be issued:

```
"constraints.ccel", line 100: TypeParametersOnly violated:
  t = BoundedArray ("BArray.h", line 15)
  p = size ("BArray.h", line 15)
```

CCEL allows this default message format to be overridden by a programmer-defined format on a per-constraint basis. Details are available in the CCEL language specification [8].

3 A Software Architecture

The architecture for our prototype constraint-checking environment is shown in Figure 2. CCEL constraints may be specified in one or more files and/or within a C++ program in the form of specially formatted comments. All features of CCEL may be used inside C++ source, but we expect programmers will use this capability primarily to specify constraints specific to a class or file, i.e. to associate a constraint with the C++ source to which it applies. For example, a constraint stating that all subclasses of a class must redefine a particular member function might be best put in the C++ source file for the class so that programmers know that they will need to define that member function. A more generic constraint, such as that every class name must begin with an upper case letter, might go in a file containing nothing but stylistic constraints.

The constraints specified in CCEL constraint files and the constraints specified in special comments in C++ source code together comprise the set of applicable constraints for the software system to be checked. This set of constraints is then used as input to a constraint checker, which employs the services of an "oracle" about the C++ system being checked. The oracle is in essence a virtual database system containing information about C++ programs. There are many actual database systems containing such information (e.g., Reprise[20], CIA++[7], and XREFDB[11]), and our virtual database interface allows us to decouple the constraint-checker from any particular database. In fact, our eventual virtual database interface will be OQL ("Object Query Language") [19], a virtual interface to *many* database systems.

The output of the constraint checker is a series of violation messages. These may be viewed as is, or they may be parsed by higher-level tools, such as Emacs [21] or the annotation editor inside FIELD [18]. Use of such tools allows programmers to see not only CCEL constraint violation diagnostics, but also the CCEL source giving rise to the violation and the C++ source that violates the constraint. This makes it much easier to locate and eliminate constraint violations.

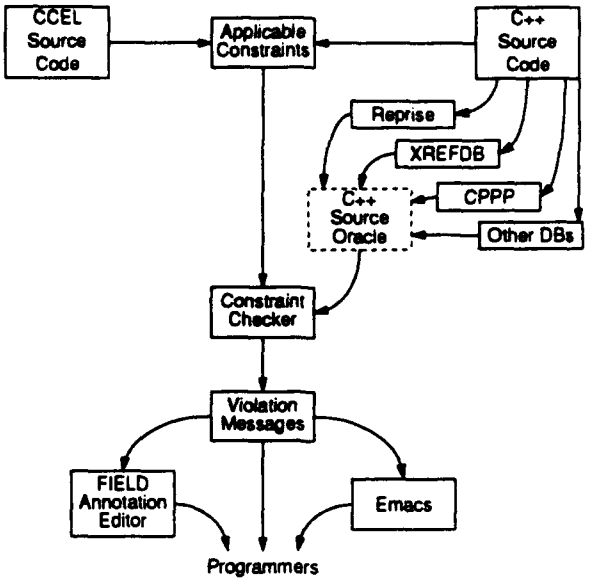


Figure 2: CCEL System Architecture

Our earlier working paper [4] described a prototype implementation of CCEL that was less elaborate than that shown in Figure 2. We are currently implementing a second-generation prototype that corresponds to the current (much expanded) language specification [8] and that is based on the database XREFDB.

4 Application to Other Languages

The classes supported by CCEL and the member functions of those classes are clearly specific to C++. The more fundamental design principles behind CCEL, however, apply equally well to other object-oriented languages. The kinds of constraints described in Section 1 exist for languages like Smalltalk and CLOS as much as they do for C++ [9]. The desireability of choosing a syntax, semantics, and conceptual model that is familiar to programmers is as important for an Eiffel constraint language as it is for CCEL. The software engineering considerations that allow CCEL constraints to be bundled into constraint classes, to be explicitly disabled, and to have user-specified scopes and violation messages are as important for Object Pascal programmers as they are for C++ software developers. Furthermore, the software architecture depicted in Figure 2 contains nothing that tailors it to the idiosyncrasies of C++. In short, the primary design considerations — both in terms of the language itself and the implementation of that language — are divorced from the specifics of C++ and can be directly applied to constraint languages for other object-oriented languages.

5 Related Work

In their analysis of the CLOS Metaobject Protocol [9], Kiczales and Lamping identified a number of issues germane to the design of extensible class libraries, and they proposed a set of informal techniques by which to specify requirements and restrictions on classes inheriting from the library. CCEL is an important step towards formalizing such requirements and restrictions and toward making them amenable to automatic verification.

Support for formal design constraints in the form of assertions or annotations was designed into Eiffel [14], has been grafted onto Ada in the language Anna [13], and has been proposed for C++ in the form of A++ [3, 2]. This work, however, has grown out of the theory of abstract data types [12], and has tended to limit itself to formally specifying the semantics of individual functions and/or collections of functions (e.g., how the member functions within a class relate to one another). In general, violations of these kinds of

constraints can only be detected at runtime. Our work on CCEL has a different focus. We are interested in constraints whose violations can be detected at compile time, and we are further interested in addressing the need to constrain relationships between classes, which Eiffel, A++, and Anna are unable to do. CCEL can also express constraints on the concrete syntax of C++ source code (e.g., CCEL class-specific naming conventions); this is also outside the purview of semantics-based constraint systems.

Acknowledgements

Yueh hong Lin provided valuable comments on earlier versions of this paper.

Support for this research was provided by the NSF under grants CCR 9111507 and CCR 9113226, by ARPA order 8225, and by ONR grant N00014-91-J-4052.

References

[1] David R. Barstow, Howard E. Shrobe, and Erik Sandewall, eds., *Interactive Programming Environments.* McGraw-Hill, 1984.

[2] Marshall P. Cline and Doug Lea, "The Behavior of C++ Classes," in *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pp. 81–91, September 1990.

[3] Marshall P. Cline and Doug Lea, "Using Annotated C++," in *Proceedings of C++ at Work - '90*, pp. 65–71, September 1990.

[4] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss, "CCEL: A Metalanguage for C++," in *USENIX C++ Conference Proceedings*, August 1992. Also available as Brown University Computer Science Department Technical Report CS-92-51, October 1992.

[5] Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[6] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico, *Data Abstraction and Object-Oriented Programming in C++.* John Wiley & Sons, 1990.

[7] Judith E. Grass and Yih-Farn Chen, "The C++ Information Abstractor," in *USENIX C++ Conference Proceedings*, pp. 265–277, 1990.

[8] Yueh hong Lin and Scott Meyers, "CCEL: The C++ Constraint Expression Language." In preparation, February 1993.

[9] Gregor Kiczales and John Lamping, "Issues in the Design and Specification of Class Libraries," in *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '92)* (Andreas Paepcke, ed.), pp. 435–451, October 1992.

[10] Moises Lejter, Scott Meyers, and Steven P. Reiss, "Adding Semantic Information To C++ Development Environments," in *Proceedings of C++ at Work - '90*, pp. 103–108, September 1990.

[11] Moises Lejter, Scott Meyers, and Steven P. Reiss, "Support for Maintaining Object-Oriented Programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, December 1992. Also available as Brown University Computer Science Department Technical Report CS-91-52, August 1991. An earlier version of this paper appeared in the *Proceedings of the 1991 Conference on Software Maintenance* (CSM '91), October 1991. This paper is largely drawn from two other papers [15, 10].

[12] Barbara Liskov and John Guttag, *Abstraction and Specification in Program Development.* The MIT Press, 1986.

[13] D. Luckham, F. von Henke, B. Krieg-Bruckner, and O. Owe, *Anna, A Language for Annotating Ada Programs: Reference Manual*, vol. 260 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[14] Bertrand Meyer, *Object-Oriented Software Construction.* Prentice Hall International Series in Computer Science, Prentice Hall, 1988.

[15] Scott Meyers, "Working with Object-Oriented Programs: The View from the Trenches is Not Always Pretty," in *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pp. 51–65, September 1990.

[16] Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, 1992.

[17] Scott Meyers and Moises Lejter, "Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++," in *USENIX C++ Conference Proceedings*, pp. 29–40, April 1991. Also available as Brown University Computer Science Department Technical Report CS-91-51, August 1991.

[18] Steven P. Reiss, "Connecting Tools using Message Passing in the FIELD Program Development Environment," *IEEE Software*, pp. 57–67, July 1990. Also available as Brown University Computer Science Department Technical Report CS-88-18, "Integration Mechanisms in the FIELD Environment," October 1988.

[19] Steven P. Reiss and Manojit Sarkar, "Generating Program Abstractions." Working paper, September 1992.

[20] David S. Rosenblum and Alexander L. Wolf, "Representing Semantically Analyzed C++ Code with Reprise," in *USENIX C++ Conference Proceedings*, pp. 119–134, April 1991.

[21] Richard M. Stallman, "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," in *Proceedings of the ACM SIGPLAN/SIGOA Symposium on text Manipulation*, pp. 147–156, June 1981. Reprinted in [1, pp. 300–325].

A Examples

The CCEL constraints that follow serve to demonstrate not only the expressiveness of CCEL itself, but also the kinds of constraints that C++ programmers might well want to express.

The following two constraints are taken from Meyers' book [16]:

```
// Subclasses must never redefine an inherited non-virtual member function:
NoNonVirtualRedefines (
    Class B;
    Class D | D.is_descendant(B);
    MemberFunction B::bmf;
    MemberFunction D::dmf | dmf.overrides(bmf);
    Assert(bmf.is_virtual());
);

// The return type of operator= must be a reference to the class:
ReturnTypeOfAssignmentOp (
    Class C;
    MemberFunction C::mf | mf.name() == "operator=";
    Assert(mf.is_reference() && mf.type().basic_type() == C);
);
```

The constraint that structs in C++ should be the same as structs in C (useful for maintaining data structure compatibility between the two) consists of three separate constraints: (1) no struct may contain a non-public member, (2) no struct may contain a function member, and (3) no struct may have a base class. These constraints may be combined into a single constraint class `StructsAreSimple` as follows:

```
// Structs in C++ should be just like structs in C:
StructsAreSimple {

    Class s | s.is_struct();

    AllMembersPublic (
        DataMember s::mv;
        Assert(mv.is_public());
    );

    NoMemberFunctions (
        MemberFunction s::mf;
        Assert(FALSE);
    );

    NoBaseClasses (
        Class base | s.is_descendant(base);
        Assert(FALSE);
    );
};
```

Here is another constraint from Meyers' book, this one also employing a constraint class:

```
// All classes declaring or inheriting a pointer member must declare a
// copy constructor:
NecessaryCopyConstructors {

  // All classes declaring a pointer member must declare a copy ctor:
  PtrDeclImpliesCopyCtor (
      Class C;
      DataMember C::mv | mv.is_pointer();
      Assert(MemberFunction C::mf; Parameter mf(p); |
             mf.name() == C.name() && mf.num_params() == 1  &&
             p.type().basic_type() == C && p.is_reference() &&
             p.is_const());
  );

  // All classes inheriting a pointer member must declare a copy ctor:
  InherPtrImpliesCopyCtor (
      Class B;
      Class D | D.is_descendant(B);
      DataMember B::mv | mv.is_pointer();
      Assert(MemberFunction D::mf; Parameter mf(p); |
             mf.name() == D.name() && mf.num_params() == 1  &&
             p.type().basic_type() == D && p.is_reference() &&
             p.is_const());
  );
};
```

The following constraint enforces a common rule of style:

```
// Members must be declared in this order:  public, protected, private:
MemberDeclOrdering {

    Class  C;
    Member C::pub  | pub.is_public();
    Member C::prot | prot.is_protected();
    Member C::priv | priv.is_private();

    PublicBeforeProtected (
        Assert(pub.begin_line() < prot.begin_line());
    );

    PublicBeforePrivate (
        Assert(pub.begin_line() < priv.begin_line());
    );

    ProtectedBeforePrivate (
        Assert(prot.begin_line() < priv.begin_line());
    );
};
```

Higher-Order Logic Programming as Constraint Logic Programming

Spiro Michaylov
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277, U.S.A.
spiro@cis.ohio-state.edu

Frank Pfenning
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891, U.S.A.
fp@cs.cmu.edu

Abstract

Higher-order logic programming (HOLP) languages are particularly useful for various kinds of meta-programming and theorem proving tasks because of the logical support for variable binding via λ-abstraction. They have been used for a wide range of applications including theorem proving, programming language interpretation, type inference, compilation, and natural language parsing. Despite their utility, current language implementations have acquired a well-deserved reputation for being inefficient. In this paper we argue that HOLP languages can reasonably be viewed as Constraint Logic Programming (CLP) languages, and show how this can be expected to lead to more practical implementations by applying the known principles for the design and implementation of practical CLP systems.

1 Introduction

Higher-order logic programming (HOLP) languages [17] typically use a typed λ-calculus as their domain of computation. In the case of λProlog [18] it is the simply-typed λ-calculus, while in the case of Elf [22] it is a dependently typed λ-calculus. These languages are particularly useful for various kinds of meta-programming and theorem proving tasks because of the logical support for variable binding via λ-abstraction. They have been used for a wide range of applications including theorem proving [3], programming language interpretation [5, 13], type inference [21], compilation [6], and natural language parsing [20]. Despite their utility, current language implementations have acquired a well-deserved reputation for being inefficient. In this paper we argue that HOLP languages can reasonably be viewed as Constraint Logic Programming (CLP) languages [8]. Measurements with an instrumented Elf interpreter confirm that such a view can produce practical benefits, as the known principles for the design and implementation of practical CLP systems [9, 12] are directly applicable to making implementations of HOLP languages more efficient.

The core domain of the languages we consider is the set of typed λ-expressions, where abstraction and application are the only interpreted operations and equality is the only relation (interpreted as $\beta\eta\alpha$-convertibility). There are other features of these languages that distinguish them from the CLP language scheme as defined in [8], for example, higher-order predicates, dependent or polymorphic types, modules, embedded implication and universal quantification. Many of these features have been addressed in a satisfactory way in ongoing implementation projects at Duke and IRISA/INRIA in Rennes. Surveys and further

references to the design of these implementations can be found in [11] and [1]. In this paper we will concentrate on the issues related to the view of higher-order logic programming as constraint logic programming which, we believe, has the most fundamental impact on expected execution speed.

2 Solving Equations Between Typed λ-Expressions

Full unification in higher-order languages is clearly impractical, due to the non-existence of minimal complete sets of most-general unifiers [7]. Therefore, work on λProlog has used Huet's algorithm for *pre-unification* [7], where so-called flex-flex pairs (which are always unifiable) are maintained as constraints, rather than being incorporated in an explicit parametric form. Yet, even pre-unifiability is undecidable, and sets of most general pre-unifiers may be infinite. While undecidability has not turned out to be a severe problem, the lack of unique most general unifiers makes it difficult to accurately predict the run-time behavior of λProlog programs that attempt to take advantage of full higher-order pre-unification. It can result in thrashing when certain combinations of unification problems have to be solved by extensive backtracking. Moreover, in a straightforward implementation, common cases of unification incur a high overhead compared to first-order unification. These problems have led to a search for natural, decidable subcases of higher-order unification where most general unifiers exist. Miller [15] has suggested a syntactic restriction (L_λ) to λProlog, easily extensible to related languages [23], where most general unifiers are unique modulo $\beta\eta\alpha$-equivalence.

Miller's restriction has many attractive features. Unification is deterministic and thrashing behavior due to unification is avoided. Higher-order unification in its full power can be implemented if some additional control constructs (**when**) are available [16].

However, our empirical analysis [14] suggests that this solution is unsatisfactory, since it has a detrimental effect on programming methodology, and potentially introduces a new efficiency problem. Object-level variables are typically represented by meta-level variables, which means that object-level capture-avoiding substitution can be implemented via meta-level β-reduction. The syntactic restriction to L_λ prohibits this implementation technique, and hence a new substitution predicate must be programmed for each object language. Not only does this make programs harder to read and reason about, but a substitution predicate will be less efficient than meta-language substitution. This is not to diminish the contribution that L_λ has made to our understanding of higher-order logic programming. As we will describe below, it forms the basis for our approach to the implementation of HOLP languages.

3 A Practical Approach to Constraint Logic Programming

The generality of the CLP scheme allows languages to be defined that raise two important implementation problems:

- *High overhead for frequently-occurring simple constraints.*
  It has been observed [9, 12] that the constraints that occur most frequently in the execution of programs in many CLP systems are relatively simple. However, the generality needed to solve the more complicated, but rarely occurring constraints tends to introduce overheads for solving all constraints. Ideally, it should be possible to solve the simple constraints without incurring this overhead.

- *Some constraints may be too hard to solve or solve efficiently.*
  For many seemingly desirable domains, the required decision algorithms simply do not exist. For others, the decision problem may be open. For many more domains, either the decision problem is known to be intractable, or the best known algorithms are impractical. Even when a reasonably efficient decision procedure exists, it may be incompatible with the CLP operational model. In particular, a CLP implementation requires incremental satisfiability testing to be efficient. That is, it must be possible to determine efficiently whether a satisfiable set of constraints, augmented with a new constraint, is still satisfiable. Efficiency here loosely means that the time should be proportional more to the size of the added constraint than that of the previous, satisfiable, set. Furthermore, because of backtracking, it must be possible to undo such augmentations of the constraint set efficiently.

Solving the first problem requires that the system be implemented with a bias towards frequently occurring constraints. A data structure that is most appropriate for certain special cases but cumbersome and inefficient for the general case can often result in dramatically improved overall performance.

One approach to the second problem is to syntactically restrict the kinds of constraints on the given domain that can be expressed. Such syntactic restrictions determine what expressions can be constructed using the operators, and what expressions the various relation symbols can be applied to. For example, arithmetic expressions could be restricted to be linear. It turns out that syntactic restrictions on constraints often rule out useful and natural programs. Such programs contain syntactically complex expressions that are typically simplified by the time they are selected at runtime. For example, a non-linear expression could become linear after instantiation of some variables.

We advocate another approach, by which constraints that cannot be decided (or decided efficiently) at the time they arise are delayed with the expectation that the problem will be simplified under additional constraints. This approach can be justified under two diametrically opposed philosophies underlying constraint logic programming.

In the first, perhaps more traditional view of constraint programming, it is important that the programmer should not have to be concerned with when information becomes available but just needs to provide *enough* constraints for it *eventually* to become available. Under this philosophy, the delaying approach achieves some amount of independence of the order in which constraints arrive at the solver without unduly restricting programs. This philosophy and the justification of delay is described in considerable detail by Jaffar *et al.* in [10], where it is argued further that delay does not require the programmer to think substantially more algorithmically.

The second view advances that a constraint logic programming language is a logic with a completely specified operational semantics, which programmers should know in order to predict runtime behavior and evaluate the efficiency of their programs. Under this view, the delaying semantics is simply a design decision in the specification of the language permitting a larger range of algorithms to be expressed concisely and naturally.

It is shown in [10] that delay can be implemented with overhead proportional to the number of delayed constraints whose state is affected by each additional constraint, rather than the total number of delayed constraints. The issue of how to restrict a CLP language appropriately has often arisen and been addressed in different ways in real systems. In CLP($\mathcal{R}$) [9], the selection rule is modified to delay nonlinear constraints until they become linear. A similar approach to nonlinear arithmetic has been adopted in recent commercial versions of Prolog III. Furthermore, the same approach is now used in Prolog III to deal with the intractability of word unification: word equations are delayed until the length of the value of initial variables is known.

We have studied a selection of 12 representative and non-trivial Elf programs with a total of about 3500 lines of code [14]. We analyzed these programs from a static and dynamic perspective. Our study demonstrates that the above observations and strategies for dealing with the problems of CLP languages in general are directly applicable to HOLP languages, and that a considerable performance improvement can be expected. Conversely, the HOLP languages provide further evidence of the general applicability of this approach.

4 Special Cases of HOLP Constraints

Terms in Elf and λProlog that contain no abstraction or functional variables correspond directly to first-order terms (as in Prolog). Our empirical study showed that most unification was either simple assignment or first-order (Herbrand) unification: around 95%, averaged over all examples. When λ-abstractions are present, substitution of a term for a bound variable (β-reduction) is a common operation. Most of these (about 95%) substitute a parameter[1] for a variable. Because first-order unification and parameter substitution dominate, the representation should be designed to handle these cases particularly efficiently.

The obvious representation for terms is that corresponding to first-order abstract syntax: a DAG with special nodes for application, abstraction etc. This is problematic because the frequently occurring first-order unification cases rely heavily on finding the principal functor, which in this representation is at the head

[1] A parameter (sometimes called *eigenvariable*) acts like a constant in unification, but has proper scope. It arises from solving universally quantified goals.

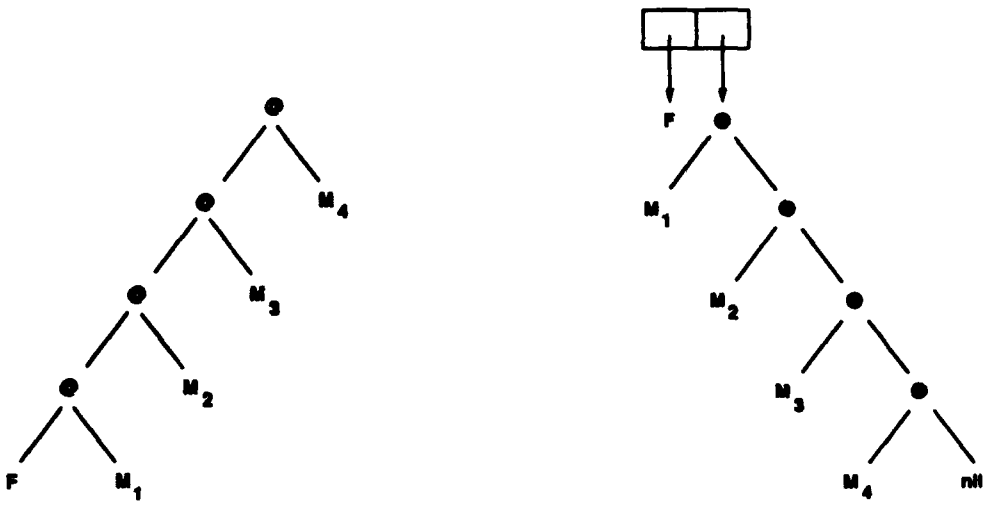


Figure 1: Conventional and Functor/Arguments term representations for Elf

of the spine of applications. Consider, for example, the representation of the term $FM_1M_2M_3M_4$ shown on the left in Figure 1. The major inefficiency results when a disagreement pair must be classified. The classifications are mostly based on the nature of the head and in particular on whether it is a constant or not. Furthermore, in the frequently occurring Rigid-Rigid case (where both heads are parameters or constants), it is necessary to know *which* constant: the pair is only decomposed into argument pairs if the heads are identical; otherwise unification fails. In the left-associative representation, obtaining information about the head is too expensive. This suggests a representation as a pair of a functor and a list of arguments. This representation, for the same example, is shown on the right in Figure 1. Notice that this representation also makes it easier to make use of clause indexing on rigid term heads.

This Prolog-like functor/argument representation can be problematic when the head is a variable, since it may be bound to an expression which requires some normalization, producing a new head, and the old one needs to be stored for backtracking in some way. These complications are outbalanced by the efficiency improvement for the simple cases, since the overwhelming majority of λProlog or Elf equality constraints can be solved by Prolog unification. We conclude that functor/argument representation is an essential optimization for a λProlog or Elf implementation.[2]

In principle, an important part of term comparison in these languages is the test for α-convertibility. The Duke representation proposal [19] suggests a de Bruijn representation [2] of terms for this reason. While that suggestion may well be appropriate, the empirical study showed the comparison of two abstractions to be a rare occurrence, and so this consideration alone should probably not be allowed to determine the choice of term representation.

Similarly, the implementation of substitution is an important issue. The dominant kinds of substitutions are those that replace bound variables by parameters—the representation should be optimized towards handling this case efficiently. This is a much more difficult issue than the representation of application, but our observations suggest that an explicit way to shift variable references in a term (through a special form of environment) might solve the efficiency problem associated with parameter substitutions.

[2]Such a representation was in fact used pervasively in Nadathur's original implementation of λProlog up to LP2.7. In the current Elf implementation, a functor/argument representation is used as an intermediate form in the constraint solver for the reasons cited above.

5 Hard Constraints in HOLP

Even a brief examination of Elf and λProlog programs shows that syntactic restriction to L_λ would affect a significant proportion of programs. While these programs can be rewritten to conform to the L_λ restriction, doing so makes them harder to reason about and, with present implementation technology, significantly less efficient. Furthermore, most programs dynamically conform to the L_λ restriction even without delay, and we are aware of only one useful program that does not run properly when hard constraints are delayed [4]. On the other hand, there are programs that run significantly more efficiently when hard constraints are delayed (for example, type inference in the polymorphic λ-calculus [21]).

The operational semantics of Elf, in contrast to λProlog, is based on solving all dynamically arising equations that lie within an appropriate extension of L_λ to dependent types. All other equations (solvable or not) are delayed. We found that this addresses the problems with higher-order unification without compromising programming methodology. The primary disadvantage of this approach is that one must take care in interpreting the final answer, if it contains delayed constraints, as a conditional: each solution of the remaining constraints yields a proof of the original query. In this section, we state precisely which constraints are deemed to be hard in λProlog and Elf and how they arise, and show how the methodology described in [10] can be used to manage hard constraints in this context.

5.1 Classification of Higher-Order Terms

In Prolog each term can be classified as either a variable, a constant, or a compound term. Solving constraints over higher-order terms requires a finer classification. For example, a term might be a λ-abstraction or a β-redex. The critical cases, however, arise when the head of a term is either a variable or a constant. In our terminology, an *Evar* is an existential variable (logic variable, in Prolog terminology) and a *Uvar* is a parameter (a constant with a well-defined scope introduced when solving a universally quantified goal). An Evar E is said to *depend* on a Uvar x in a goal if E is introduced into the computation within the scope of x. If E depends on x the substitution term for E may contain occurrences of x, otherwise it may not. Terms are then classified as follows.

- **Gvar**
  $Fx_1x_2\cdots x_n, n \geq 0$ where F is an Evar, the x_i are Uvars, F does not depend on any x_i, and the x_i are all distinct.
- **Flex**
  $FM_1M_2\cdots M_n, n \geq 0$ where F is an Evar and the M_i are terms, and the Gvar conditions above are not satisfied.
- **Rigid**
  $fM_1M_2\cdots M_n, n \geq 0$ where f is a constant or a Uvar and the M_i are arbitrary terms.

Note that, to simplify the discussion, types and other classes of disagreement pairs have been omitted, since they are dealt with by straightforward recursive unifications. We should also emphasize that in this discussion the Flex case does not include the Gvar case, unlike in Huet's usage.

5.2 Classification of Constraints

A HOLP system must solve the nine kinds of equations arising from these three kinds of terms, collapsing to six kinds due to symmetry. The table in Figure 2 shows which pairs are directly solved and which are delayed. Note that the disagreement pairs that are directly solved will either have a most general solution or no solution.

The constraint solver state consists of a set of substitutions of the form $X = M$ where X is an Evar and M is a term, such that X does not occur elsewhere, and a set of Flex-Flex pairs. This is an implicit solved form. In the implementation, the Evar-Term pairs are not represented as pairs, but by pointers from the variable to its instantiation term (as in Prolog). However, Flex-Flex pairs must be represented explicitly. In addition, the constraint solver must handle hard constraints, which arise in two ways:

Core Elf Unification Table			
	Gvar	Flex	Rigid
Gvar	unify		
Flex	delay	delay	
Rigid	unify	delay	unify

Figure 2: Core unification table for Elf

1. When a new disagreement pair is a Flex-Rigid pair (under the current substitution). This corresponds to the typical source of hard constraints in languages like CLP($\mathcal{R}$).
2. When additional substitutions are added to the solver as a result of solving a new disagreement pair, and these can be used to rewrite some Flex-Flex pair already in the solver to a pair that is a hard constraint.

The second situation is unusual for constraint languages, since a conjunction of directly solvable constraints may be simplified into a hard constraint. However, this is not problematic in the context of the methodology described in [10]: it merely requires that Flex-Flex pairs be treated as if they were hard constraints when designing the wakeup system, as described below.

5.3 A Wakeup System

In this section, our aim is to describe the management of hard constraints in Elf in terms of the framework developed by Jaffar *et al.* in [10].

We need five wakeup degrees in addition to *awakened*. These are for Flex-Flex, Flex-Rigid, Rigid-Flex, Flex-Gvar, and Gvar-Flex. We note that it is not desirable to combine symmetric cases, because the transitions of the two sides of the equation depend on the binding of different variables.

The transitions between these three forms of expressions that we need to consider are as follows. Note that we do not consider leading abstractions.

1. *Flex* $\Rightarrow$ *Rigid*
   The head F in $FM_1M_2\cdots M_n$ is bound to
   $$\lambda x_1 \cdots \lambda x_k.\, gN_1 \cdots N_m$$
   where g is a constant or a Uvar and the N_i are arbitrary terms. The resulting Rigid term will be of the form
   $$gP_1 \cdots P_l.$$
2. *Gvar* $\Rightarrow$ *Rigid*
   Same as *Flex* $\Rightarrow$ *Rigid*.
3. *Flex* $\Rightarrow$ *Gvar*
   (a) All of the arguments are bound to universal variables, such that the Gvar criteria now hold. (This is very unlikely, and expensive to check for, so it has not been implemented to date).
   (b) The head F of $FM_1M_2\cdots M_n$ is bound to
   $$\lambda x_1 \cdots \lambda x_k.\, Gy_1 \cdots y_m$$
   where G is an existential variable, each y_j is either a Uvar or one of the x_i, and the resulting term is a Gvar, that is, a term of the form
   $$Gz_1 \cdots z_l$$
   such that the z_i are all distinct Uvars, and G does not depend on any of them.

4. *Gvar* $\Rightarrow$ *Flex*
The head F of $Fx_1x_2\cdots x_m$ is bound to

$$\lambda x_1 \cdots \lambda x_k.\, GN_1 \cdots N_m$$

where G is a an existential variable and the N_i are terms, such that the Gvar criteria are now violated. The resulting Flex term will be of the form

$$GP_1 \cdots P_l.$$

Notice that the above transitions admit the possibility of cycles: A Flex term can turn into a Gvar term when more information becomes available, and with still more information may turn back into a Flex term, all without backtracking. This makes the wakeup system cyclic, as shown in Figure 3. The two arcs shown using a thinner line correspond to the case that is expensive, unlikely, and omitted in the current implementation. We describe the generic wakeup conditions in symmetric pairs, to avoid notational clutter, and ignore the term that does not change in each pair.

6 Conclusion

Higher-Order Logic Programming languages differ substantially from other Constraint Logic Programming languages. However, our empirical evidence shows that the language design and implementation strategies that have made such a substantial difference to better known CLP languages are applicable here as well.

We believe that the main challenge in the design of an abstract machine and a compiler for HOLP languages that achieves Prolog's efficiency on Prolog-like programs is the design of a representation that permits efficient substitution of parameters for bound variables without incurring an undue overhead for the usual first-order unification computation.

References

[1] Pascal Brisset and Olivier Ridoux. The architecture of an implementation of λProlog: Prolog/mali. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 195–200, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

[2] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

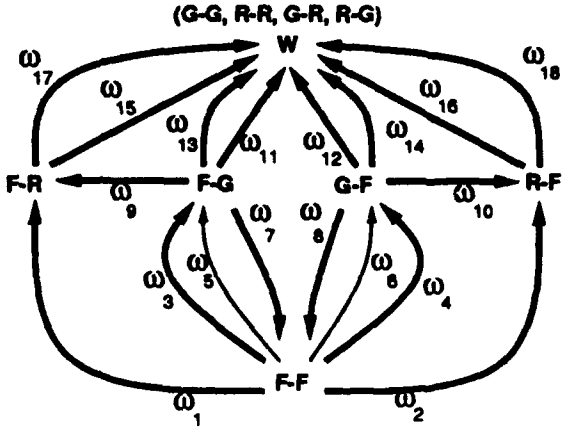
[3] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989. Available as Technical Report MS-CIS-87-109.

[4] Juergen Haas and Bharat Jayaraman. Interactive synthesis of definite-clause grammars. In Krzystof Apt, editor, *Proc. Joint International Conference and Symposium on Logic Programming*, pages 541–555, Washington, DC, November 1992. MIT Press.

[5] John Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs*. PhD thesis, University of Pennsylvania, January 1991. Available as technical report MS-CIS-91-09.

[6] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[7] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.



F	$FM_1M_2\cdots M_n$
G	$Fx_1x_2\cdots x_m$
R	$fM_1M_2\cdots M_n$

F -> R	$(\omega_1, \omega_2, \omega_{13}, \omega_{14}, \omega_{15}, \omega_{16})$	$F = \lambda x_1 \cdots \lambda x_k.\, gN_1 \cdots N_m$
F -> G	(ω_5, ω_6)	$M_1 = x_1 \wedge \cdots \wedge M_n = x_n$ $\wedge$ $Gvar(Fx_1 \cdots x_n)$
F -> G	$(\omega_3, \omega_4, \omega_{11}, \omega_{12}, \omega_{17}, \omega_{18})$	$F = \lambda x_1 \cdots \lambda x_k.\, Gy_1 \cdots y_m$ $\wedge$ $(\lambda x_1 \cdots \lambda x_k.\, Gy_1 \cdots y_m)M_1 \cdots M_n \rightarrow M'$ $\wedge$ $Gvar(M')$
G -> R	(ω_9, ω_{10})	$F = \lambda x_1 \cdots \lambda x_k.\, gN_1 \cdots N_m$
G -> F	(ω_7, ω_8)	$F = \lambda x_1 \cdots \lambda x_k.\, GN_1 \cdots N_m$ $\wedge$ $(\lambda x_1 \cdots \lambda x_k.\, GN_1 \cdots N_m)M_1 \cdots M_n \rightarrow M'$ $\wedge$ $\neg Gvar(M')$

Figure 3: Wakeup system for Elf

[8] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL), Munich, Germany*, pages 111–119. ACM, January 1987.

[9] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, July 1992.

[10] Joxan Jaffar, Spiro Michaylov, and Roland Yap. A methodology for managing hard constraints in CLP systems. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 306–316, Toronto, Canada, June 1991.

[11] Keehang Kwon and Gopalan Nadathur. An instruction set for higher-order hereditary harrop formulas (extended abstract). In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 195–200, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

[12] Spiro Michaylov. *Design and Implementation of Practical Constraint Logic Programming Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, August 1992. Available as technical report CMU-CS-92-168.

[13] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNCS/LNAI 595.

[14] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Proc. Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, PA, USA, July/August 1992. Appears as University of Pennsylvania technical report MS-CIS-92-86.

[15] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop*, pages 253–281, Tübingen FRG, December 1991. Springer-Verlag LNCS 475.

[16] Dale Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 255–269. MIT Press, July 1991.

[17] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 448–462, London, July 1986. Springer Verlag LNCS 225.

[18] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Seattle, WA, August 1988. MIT Press.

[19] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of the 1990 Conference on Lisp and Functional Programming*, pages 341–348, Nice, France, June 1990. ACM Press.

[20] Fernando C. N. Pereira. Semantic interpretation as higher-order deduction. In *Proceedings of the Second European Workshop on Logics and AI*, pages 78–96. Springer-Verlag LNCS 478, September 1990.

[21] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM Press.

[22] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[23] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

Constraint Satisfaction, Constraint Programming, and Concurrency

Ugo Montanari and Francesca Rossi

University of Pisa
Computer Science Department
Corso Italia 40, 56100 Pisa, Italy
{ugo,rossi}@di.unipi.it

Abstract

Recently constraint satisfaction has been embedded in various high-level declarative programming environments, like the Constraint Logic Programming framework, and even more recently such environments have been extended with concurrency, like in the Concurrent Constraint Programming paradigm. The merging of different areas of research is always an exciting event, and prolific of new ideas. We believe that here it is particularly so, since the above areas, while apparently very far in aims and techniques, are found more and more interconnected, to the point that their coexistence leads to new unexpected directions of research. In fact, we believe that interesting results in constraint satisfaction and in constraint programming can be mixed with concurrency with the consequence that more concurrency, as well as a more natural view of constraint programming, are derived.

1 Organization of the paper

In Section 2 we describe our general scheme for local consistency techniques for finite domain constraint problems, together with a specific efficient instance of the scheme. Then in Section 3 we show how to embed such an instance within the Constraint Logic Programming framework and we discuss its efficiency. In Section 4 we define, in terms of graph rewriting, a concurrent abstract machine for Concurrent Constraint programs, and we describe how to obtain a suitable semantics which is able to show all the concurrency and the nondeterminism contained in a program. Moreover, we show how to exploit it to derive useful information about the causal dependency of the objects involved in the computations, and we give an example of its application to the automatic parallelization of CLP programs. Finally, we give some suggestions for further research.

The presentation is rather informal. Due to the lack of space, we chose to convey the main motivations, ideas, and results instead of writing the technical development. However, the formal details can be found in [Ros93] and in the various papers cited throughout the text.

2 Constraint Satisfaction

Constraint satisfaction problems are very useful for describing many real-life problems, like scene labelling, graph coloring, VLSI routing, hardware design, software specification and design, and operation research problems. In particular, finite domain constraint problems are significantly easier to treat than general ones, and nevertheless are sufficiently descriptive of many real situations. Therefore the growing interest in such problems is widely justified.

A (finite domain) constraint problem can be specified by giving a set of variables and a set of constraints, where each constraint connects only a subset of the variables and is defined by a set of tuples of domain

values. Note that the domains of the variables are not specified separately, but can just be seen as unary constraints. Note also that a constraint may in general involve any number of variables (see for instance [MR88]), not only one or two as in most of the literature ([Mon74, Mac77, MF85, Fre78, DP88]).

Then, the solution of a constraint problem is the set of all the assignments of values to *some* of the variables which satisfy all the constraints. The reason why we depart from the usual definition of solution, which involves all the variables instead of only some of them, is that we believe it is very reasonable to let the user specify the objects in which he/she is interested, while being able to use many other objects to describe a constraint problem in detail. This is also useful when a constraint problem has to be used by many users, who may be interested to different sets of objects and do not want to know the admissible values for all the other objects.

Given this syntactic description of a constraint problem, there may be many others which represent the same problem. In particular, there may be others which are more compact, i.e. with less tuples. In fact, some of the tuples describing a constraint could be incompatible with some other constraint, and therefore could be eliminated from the syntax of the problem without changing its semantics (which is the set of all its solutions). In this sense, such tuples are redundant.

Assuming that a constraint problem is being solved by some form of a backtracking procedure, the removal of redundant tuples is particularly interesting, since it could cancel some failing branches from the search tree and thus improve the overall behavior of the search. To decide whether a tuple is redundant, it often (not always!) suffices to look only at a small portion of the problem.

Many algorithms have been developed to remove redundant tuples. They all have the same aim, but they differ in their power, i.e. in the amount of redundant tuples they are able to remove. In this line we find, in order of increasing power, the arc-consistency algorithm ([Mac77, MF85]), the path-consistency algorithm (described for the first time in a pioneering paper by one of the authors [Mon74]), the k-consistency framework ([Fre78]), and the adaptive-consistency algorithm ([DP88]). A survey containing a description of most of them can be found in [Kum91]. All such algorithms, usually called local consistency algorithms because they work at a local level, have a polynomial complexity, but are complete (i.e., they remove all the redundant tuples) only when applied to specific classes of problems ([Fre88]). When they are complete, the resulting backtracking search never backtracks, or has a bounded amount of backtracking.

2.1 A general scheme

Even though most the local consistency algorithms have been developed quite separately and without a common notation nor a uniform way of specifying their properties, it is possible to recognize that they are quite similar at the core, in that they all try to remove redundancy at some local level. Based on this observation, we have developed a general scheme which embeds all the previously proposed local consistency algorithms, as well as some new ones, as specific instances ([MR91a]).

In general, the significance of a scheme lies in the fact that all the properties which hold for the scheme are automatically inherited by all its instances. In our case, this was extremely true and convenient, since there were properties which had never been stated formally before for the single algorithms. In fact, while termination of a local consistency algorithm has always been one of the concerns, and therefore it is possible to find several proofs of such property in the various papers introducing new algorithms, the fact that the resulting constraint problem is unique has always been treated very informally. Instead, in our scheme both these properties have a formal proof, which may then be inherited by all the algorithms.

In the scheme, each concrete algorithm is specified once a specific set of basic operators, and a strategy for their application, is chosen. Each of these basic operators is very simple, and involves just the analysis of a subproblem of small size and the removal of some of its redundant tuples. An algorithm consists then of the application of such operators, in the order given by the strategy, until stability is reached (i.e., until no more tuples can be removed by any of the operators). Note that in general an operator may have to be applied more than once before stability is reached, since the application of another operator may make some more tuples in the subproblem associated to the first one to be recognized as redundant only at a later application.

For example, arc-consistency (resp., path-consistency, k-consistency) can be achieved by choosing, as the set of basic operators, all the subproblems spanned by any two (resp., three, k) variables. Note, however, that the scheme is more general, since there is no requirement that all the operators represent subproblems

of the same size.

If we consider the partial order consisting of all the constraint problems equivalent to the given one but with possibly less tuples, ordered by tuple set inclusion, then each basic operator is just a closure operator (i.e., an extensive, monotone, and idempotent function) over such partial order, and thus a local consistency algorithm is simply the application of all these closure operators until quiescence. This observation will turn out to be very important when considering possible linguistic supports for specifying constraint problems and local consistency algorithms, since an adequate support would be one where closure operators are first class objects.

2.2 A Specific Instance: Perfect Relaxation

As noted above, most of the local consistency algorithms developed in the past seem to have the recurring property that all the considered subproblems (where redundancy has to be removed) have the same size (for one algorithm). An exception is the adaptive-consistency algorithm ([DP88]). This is obviously an unreasonable restriction, which may prevent the development of new and convenient algorithms. In our scheme, there is no such restriction about the basic operators, which may be very different in size.

An example of a new algorithm which is an instance of the scheme but which may involve very different basic operators is called perfect relaxation ([MR86, RM89]). The main feature of such algorithm is that it is always complete, and that it uses each of the basic operators exactly once. Moreover, it is very efficient when applied to particular classes of constraint problems. Informally, such problems can be described as consisting of a set of small subproblems loosely connected among them in a tree-like shape. For such classes, perfect relaxation is linear in the size of the solved problem.

More precisely, consider a finite set I of finite domain constraint problems, and consider the class of all those problems which can be built by putting together the problems in I in a tree-like shape. This class is called I-structured. Perfect relaxation is able to solve efficiently any problem in an I-structured class. Since such problem consists of a tree of subproblems belonging to I, the idea is to solve each subproblem in the tree in a bottom-up fashion, from the leaves to the root. In terms of the local consistency scheme, this amounts to having the subproblems in I as basic operators, and the bottom-up traversal as the strategy.

It is possible to show that this bottom-up procedure obtains the complete solution of the problem. Since the subproblems have bounded size (it is bounded once the set I is given), the whole solution process involves as many steps as the subproblems, which can be thought as in number linear with the size of the problem (each step may be exponential in the size of a subproblem and thus is a constant since such size is bounded). Therefore the whole complexity is linear in the size of the problem to be solved.

However, and not surprisingly, not all classes of problems can be recast as I-structured classes. For example, there is no I such that the class of rectangular lattices is I-structured.

3 Constraint Satisfaction in CLP

A lot of effort is currently being put into the task of embedding constraint satisfaction facilities within high level programming languages. In doing so, the aims are many: to be able to treat more complex constraint problems, specifying them in a hierarchical or even recursive way, to combine different algorithms and/or constraint domains, to parametrize constraint solving, to incrementally generate constraint problems, and even to exploit constraints in tasks which have never been considered before, like the exchange of partial information between concurrent program agents.

In particular, one of the most recent languages which are based on constraints is the Constraint Logic Programming (CLP) scheme ([JL87]). It is an extension of logic programming ([Llo87]) where term equalities are replaced by arbitrary constraints and unification by an arbitrary constraint solving algorithm. It is a scheme because it is parametric w.r.t. an underlying constraint system, which is a collection of algorithms to handle and solve a specific class of constraints.

The naive approach to the construction of an instance of the CLP scheme, i.e., a specific CLP language, consisting of the choice of a specific class of constraints, and of the embedding of any efficient algorithm for such class within CLP, can be sometimes reasonably satisfactory. However, issues of incrementality (i.e., the ability of exploiting the knowledge that the previous set of constraints is solvable in order to not repeat

the work done in previous steps) and canonicality (i.e., the existence of a canonical form for each constraint set, which is compact and easy to use) have always to be considered. But most of all, such naive approach does not exploit one of the fundamental features of CLP, which consists of the monotonic accumulation of constraints, performed by the addition of a small constraint set at each computation step.

Perfect relaxation can instead take advantage of such feature in order to check efficiently the satisfiability of the accumulated constraints ([MR91c, MR92b, MR91b]). In fact, consider the instance of the CLP framework over finite domain constraints, called CLP(FD). Then, each computation step in CLP(FD) involves the expansion of a selected subgoal in the current goal via a program clause, together with the consistency check of a set of constraints (those already in the goal plus those in the program clause). Now, the important observation is that this set of constraints can always be seen as an I-structured problem. In fact, we can think of the constraints added at each computation step as the problems of the set I. Moreover, due to the way computation steps are performed (mainly, by letting the current goal and the newly added goals and constraints to interact only via the clause head), we never obtain a cyclic structure, but always a tree whose nodes are the constraints added at each step. This means that perfect relaxation can be used to perform the consistency check required at each derivation step.

Let us now consider the complexity of such use of perfect relaxation. If the constraint set has been obtained via n computation steps, then perfect relaxation needs n steps as well, and each of such steps is exponential in the size of the constraint subproblem added at the corresponding computation step. Therefore, by the properties of perfect relaxation, if such size is bounded, then perfect relaxation is linear in the size of the whole set of constraints.

Unfortunately, given a CLP(FD) program, it is not possible in general to find a bound to the size of such set I of the possible constraint sets added at a derivation step in any of the possible executions of the program. This derives from the fact that in general a CLP(FD) program, although dealing only with finite domain constraints, may contain some data constructors in those atoms which are not constraints. In fact, any CLP instance is always able to handle term unification, no matter which constraint domain is supposed to deal with. In some sense, we may say that a CLP language has two different constraint solvers, one of which is always the unification of Herbrand terms.

The presence of terms in program clauses, for instance of lists, makes it impossible in general to derive a constant bound to the size of the constraints which are going to be added at a derivation step, since the number of variables potentially involved may arbitrarily grow in a computation. However, there are special cases when this cannot happen. The simplest of them is the case where CLP(FD) coincides with Datalog, which means that no data constructor is present. More interesting and general cases, where functions are allowed but restrictions are posed on the structure of the clauses and/or of the goals, can be studied, for example via abstract interpretation of the CLP(FD) programs, and suitably characterized.

It is important to notice that, while for many algorithms it may be difficult to produce an incremental version, for perfect relaxation it is very easy and convenient. In fact, the set of constraints added in the current derivation step is always a leaf of the tree structure. Therefore the algorithm, which is, we recall, just a bottom-up traversal of the tree, may simply start its traversal from this leaf, and then continue in the path from such leaf to the root, until no change is made in one of the steps (i.e., no redundant tuple is removed). Therefore, while the solution of an entire constraint problem from scratch needs the traversal of the whole tree, the fact that in CLP constraints are generated incrementally makes perfect relaxation even more efficient, since each derivation step requires only the traversal of at most a path from one leaf to the root. For balances trees, the complexity is thus logaritmic in the size of the tree and of the whole constraint set.

4 From CLP to CCP: a Concurrent Abstract Machine for CCP

In the CLP framework, the underlying constraint solver is treated as a black box, which cannot be changed or even looked at by the user. Moreover, the user cannot extend such constraint handler at the program level, because the language does not provide the necessary operations on constraints which are usually used by the constraint solution algorithms.

Although first steps towards the extension of CLP via user-written parts of the underlying constraint handler have been recently proposed ([Fru92]), there is already a language framework which seems to have the

desired flexibility which is instead lacking in CLP. It is the concurrent constraint (cc) programming framework ([Sar89, SR90a]), which can be thought of as CLP plus concurrency. More precisely, a cc program consists of a set of concurrent agents which share a set of variables that are subject to some constraints. Each agent may then either add (*tell*) a new constraint to the current set of constraints, or check (*ask*) whether a new constraint is logically entailed by the current constraint set, or split into more agents. As CLP, also the cc framework is parametric w.r.t. the underlying constraint system, which can be described in a simple way as a set of primitive constraints (tokens) plus an entailment relation relating tokens to tokens, and stating when a token is entailed by some other tokens. Starting from an initial constraint and an initial agent (which is in some sense a query), the computation evolves through a monotonic accumulation of constraints until quiescence, at which point the current set of constraints is the answer to the initial query.

It is important to notice that the ask operation is intimately connected to the presence of a concurrent model of computation. In fact, an agent asking a certain constraint may be suspended (if that constraint is not entailed by the current constraint set but it is consistent with it), waiting for another agent to add enough information (i.e., constraints) so that that constraint become either entailed or inconsistent.

4.1 Local consistency in CCP

In our view, one of the most important aspects of the entailment operation is that most constraint solution (or local consistency) algorithms are based exactly on such two operations: consistency and entailment. as a consequence, these algorithms can be easily specified as cc programs. This can be checked also by considering that any cc agent can be assimilated to a closure operator ([SR90b]), just like the basic operators of the constraint solution algorithms (as noted above). More informally, it is enough to see that each basic operator of a local consistency algorithm can be cast as a cc agent which asks whether a certain tuple is entailed by the other constraints of the subproblem (which means that it is consistent with them), and, if so, then tells it as a new constraint.

Among the local consistency algorithms, perfect relaxation is here able to exploit more effectively, as it is in CLP, the incremental nature of constraint accumulation in the cc framework. In fact, both the tell operation, which requires to test whether the told constraint is consistent with the current constraint set (and therefore it is assimilable to a derivation step in CLP), and the ask operation, which requires the test for entailment, can efficiently and naturally be performed via perfect relaxation. The coexistence of these two operations, which are so intimately connected to each other, allows to reduce even more the complexity of each application of the perfect relaxation algorithm. In fact, the knowledge obtained from performing one of such operations could be maintained in suitable data structures and used to speed up the performing of the other operation.

This uniformity of basic local consistency operators and cc agents means that the user can naturally extend the underlying built-in constraint solver by simply writing an additional piece of program. In other words, in the cc framework program agents and constraints are homogeneous, both from a static and form a dynamic point of view, and this makes such a framework very natural and flexible in terms of constraint programming.

4.2 The abstract machine

We have developed an abstract machine for cc programs, based on graphs and graph rewriting rules, which is able to express such homogeneity in a natural way, and also to exploit it for a better understanding of the level of concurrency implicitly contained in cc programs ([MR91d, MR92c]). One of the main features of such machine is that it is concurrent, while most of the computation models used for describing the dynamic behavior of cc programs are instead sequential and thus reduce concurrency to nondeterminism (that is, they cannot distinguish between computation steps which can happen in any order but not simultaneously and computation steps which can happen in any order and also simultaneously).

At any given point, a cc computation state consists of the active agents and of the constraints already generated (i.e., the current store). Our idea is to represent such a state via a graph, where nodes represent program variables, and arcs represent either agents or primitive constraints. In this way, each agent or primitive constraint is connected to the variables it involves, and they are uniformly represented. This

uniform representation of agents and constraints seem very reasonable to us, since agents can be simply seen as user-defined constraints.

Then, each computation step involves either the evolution of an agent, which can perform one of the allowed operations (ask, tell, ...), or the evolution of some set of primitive constraints, which can cause the addition of a new constraint to the store if such new constraint is entailed by them, as specified by the entailment relation. All such operations can be uniformly described as the application of a graph rewrite rule to the graph representing the current state.

Thus, our abstract machine has graphs as states and applications of graph rewrite rules as transitions. However, an algebraic description of this machine, based on term rewriting modulo certain structural axioms representing graphs, is also possible ([CMR92a, CMR92b]).

Note that the interpretation of each pair, say $c \vdash d$, of the entailment relation $\vdash$, as the addition of constraint d to the store whenever c is already in the store, has three consequences. First, the constraint system is not seen as a black box which can passively answer consistency or entailment tests, but as an active entity of the computation environment, which is more realistic if one wants to model the entire environment and not only a part of it. Second, it has a completely distributed representation. This gives to our machine a degree of concurrency higher than usual. In fact, it increases the concurrency: i) within the constraint system; ii) between the constraint system and the agents; and iii) within the agents. Third, the ask operation is reduced to testing the presence of certain constraints in the current store (instead of their entailment).

Note again that this machine is inherently concurrent, since different rewrite rules can be applied in parallel provided that no conflict situations actually occur.

Informally, a graph rewrite rule is a local operation which, given a graph, deletes some of its items (nodes and/or arcs, or, in other words, a subgraph), generates new items, and tests other items for presence. This third set of items (those that are tested for presence) can be thought of as a *context*. In fact, they are not affected by the application of the rewrite rule, but they are needed for such application to take place.

The use of a context-dependent rewriting formalism is particularly important in describing the cc framework, since it allows for a convenient and correct representation of the asked constraints. In fact, when an agent checks the entailment of a constraint (i.e., it performs an ask operation), such constraint is not affected by the operation, but it is nevertheless needed in order for the operation to succeed.

Moreover, the fact that asked constraints are not affected by the ask operation, and the ability of formally describing this situation, has a very convenient consequence in terms of concurrency. In fact, different agents which ask for the same constraint may evolve in parallel without any conflict. This possibility is elegantly modelled in the algebraic framework for graph grammars ([Ehr78]), where different graph rewrite rules which have the same context may be applied in parallel.

Therefore our machine is able to gain concurrency from several directions: from the fact that no sequentialization of graph rewrite rules is ever assumed, from the uniform representation of agents and constraints and a non-monolithic view of the constraint system, and from the context-dependent nature of graph rewrite rules.

This conceptual graph-based abstract machine for cc programming has been the basis for a more general abstract machine for distributed systems called CHARM ([CMR92a, CMR92b]), which has been given an algebraic description and has been shown to be able to implement graph grammars, Petri nets, and cc programs. The CHARM is also related to the CHAM machine ([BB90]), which has simpler structural axioms, and which does not exhibit the same convenient treatment of context items and of shared variables. In some sense we can say that, while the CHAM machine models multiset rewriting via the chemical metaphor, the CHARM machine models term (and graph) rewriting via the graph metaphor.

4.3 The partial order semantics

Given a computation of a cc program, which is represented by a sequence of applications of graph rewrite rules, it is possible to derive a partial order among the computation steps which reflects the existing concurrency ([MR92c]). In fact, elements of the partial order which are not related by the partial order relation are concurrent and can therefore evolve in any order or even simultaneously. Note that such partial order represents not only the given computation, but also all those computations which differ from it for the order of some concurrent steps.

However, given two cc computations, they may be represented by two different partial orders, since they may differ not only for the order of some concurrent steps but also (or alternatively) for the presence or the absence of some steps. This means that such two computations derive from different choices at the various nondeterministic choice points the program may have. Therefore, in general, a cc program is represented by a set of partial orders.

This partial-order based semantics for cc programs allows us to study and possibly exploit the concurrency in each computation, but it is not able to show where the nondeterminism occurs. In fact, different partial orders have nothing in common, even though the corresponding computation classes where just differing for few steps because of a different nondeterministic choice. Therefore we extended the partial order approach and we developed a new semantics, which consists of a structure whose elements are all the steps and the items (tokens and agents) in all the computations, and where three relations are defined among these objects: dependency, concurrency, and mutual exclusion. In this way, all the partial orders are still recoverable (as those substructures where no two elements are mutually exclusive), and in addition it is possible to see how such partial orders interact due to the nondeterminism. In other words, both AND and OR parallelism is formally described and made explicit. Two versions of such semantics have been developed, one ([MR92a]) using the so-called event structures, and another one using an extension of Petri nets with context conditions, called contextual nets ([MR93]).

The causal dependency information expressed by such partial orders (or by the structure containing them) could be conveniently used in all those tasks which would gain from a formal and faithful representation of such knowledge. In particular, we have used our semantics to help an optimizer trying to automatically parallelize CLP programs. The aim is to run in parallel as many goals as possible while not losing efficiency nor answers. The idea is to view a CLP program as a cc program (as if everything could run in parallel), and to obtain the corresponding semantic structure. Then, to add in this structure some dependency links which reflect the left-to-right order in which goals are written in the clauses. At this point, all that is still concurrent can safely be run in parallel. With these technique, we are able to define a new notion of goal independence which is more relaxed than all the existing ones, and which thus allows more goals to be recognized as independent.

We are confident that this same technique, or a very similar one, could be used for a source to source transformation of cc programs, for the transformation of CLP programs into cc programs, for the derivation of an optimal sequential scheduling for cc programs, as well as for intelligent backtracking schemes.

Our semantics can still be improved. In fact, it can only represent one kind of nondeterminism (either don't care or don't know), but not both of them together. This means that for now it can be used only for languages like CLP which only employ don't know nondeterminism, or like the concurrent logic languages and the committed-choice fragment of the cc languages, which employ only don't care nondeterminism. Instead, the cc languages may have both kinds of nondeterminism (and even more complex constructs, like AKL). However we are confident that by adopting suitable variations our semantics can be extended to handle this more general situation. Moreover, our semantic structures are very concrete. That is, any two cc programs which are syntactically different originate different structures, even though they are trivially equivalent in term of their dependency pattern. Therefore we would like to define a suitable equivalence relation among structures which would make our semantics more abstract while still retaining all its useful knowledge.

References

[BB90] G. Berry and G. Boudol. The Chemical Abstract Machine. In *Proc. POPL90.* ACM, 1990.

[CMR92a] A. Corradini, U. Montanari, and F. Rossi. Charm: Concurrency and hiding in an abstract rewriting machine. In *Proc. Fifth Generation Computer Systems 1992*, 1992.

[CMR92b] A. Corradini, U. Montanari, and F. Rossi. A concurrent abstract machine for distributed systems: Charm. *Theoretical Computer Science*, 1992. To appear.

[DP88] R. Dechter and J. Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. In Kanal and Kumar, editors, *Search in Artificial Intelligence.* Springer-Verlag, 1988.

[Ehr78] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Proc. International Workshop on Graph Grammars*. Springer Verlag, LNCS 73, 1978.

[Fre78] E. Freuder. Synthesizing constraint expressions. *Communication of the ACM*, 21(11), 1978.

[Fre88] E. Freuder. Backtrack-free and backtrack-bounded search. In Kanal and Kumar, editors, *Search in Artificial Intelligence*. Springer-Verlag, 1988.

[Fru92] Thom Fruhwirth. Constraint simplification rules. Technical report, ECRC, Munich, Germany, 1992.

[JL87] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proc. POPL*. ACM, 1987.

[Kum91] V. Kumar. Algorithms for constraint satisfaction problems: a survey. Technical Report 91-28, University of Minnesota, 1991.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1987.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1), 1977.

[MF85] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985.

[Mon74] U. Montanari. Networks of constraints: Fundamental properties and application to picture processing. *Information Science*, 7, 1974.

[MR86] U. Montanari and F. Rossi. An efficient algorithm for the solution of hierachical networks of constraints. In *Proc. International Workshop on Graph Grammars and their application to Computer Science, LNCS 291*. Springer Verlag, 1986.

[MR88] U. Montanari and F. Rossi. Fundamental Properties of Networks of Constraints: A New Formulation. In: L. Kanal and V. Kumar, Eds., *Search in Artificial Intelligence*, Springer Series in Symbolic Computation, pp. 426-449, 1988.

[MR91a] U. Montanari and F. Rossi. Constraint relaxation may be perfect. *Artificial Intelligence Journal*, 48:143–170, 1991.

[MR91b] U. Montanari and F. Rossi. Finite domain constraint problems and their relationship with logic programming. In C.W. Holsapple and A. Winston, editors, *Recent Developments in Decision Support Systems*. Springer-Verlag, 1991.

[MR91c] U. Montanari and F. Rossi. Perfect relaxation in constraint logic programming. In *Proc. International Conference on Logic Programming*. MIT Press, 1991.

[MR91d] U. Montanari and F. Rossi. True concurrency in concurrent constraint programming. In *Proc. ILPS91*. MIT Press, 1991.

[MR92a] U. Montanari and F. Rossi. An event structure semantics for concurrent constraint programming. Submitted for publication, 1992.

[MR92b] U. Montanari and F. Rossi. Finite domain constraint solving and constraint logic programming. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1992.

[MR92c] U. Montanari and F. Rossi. Graph rewriting for a partial ordering semantics of concurrent constraint programming. *Theoretical Computer Science*, 1992. special issue on graph grammars, Courcelle B. and Rozenberg eds.

[MR93] U. Montanari and F. Rossi. Contextual nets. Technical report, University of Pisa, CD Department, TR 93-4, 1993.

[RM89] F. Rossi and U. Montanari. Exact solution of networks of constraints using perfect relaxation. In *Proc. International Conference on Knowledge Representation*. Morgan Kaufmann, 1989.

[Ros93] F. Rossi. *Constraints and Concurrency*. PhD thesis, University of Pisa, 1993. Technical Report TD 14/93.

[Sar89] V. A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, 1989. To appear by MIT Press, 1989 ACM dissertation award.

[SR90a] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. POPL*. ACM, 1990.

[SR90b] V. A. Saraswat and M. Rinard. Determinate constraint programming. Technical report, Xerox Palo Alto Research Center, 1990.

Programming in CLP(BNR)

William J. Older
Bell Northern Research
Computing Research Laboratory
PO Box 3511, Station C
K1Y 4H7 Ottawa, Ontario, Canada

Frédéric Benhamou
Groupe Intelligence Artificielle
Faculté des Sciences de Luminy
case 901,163, avenue de Luminy
13288 Marseille Cedex 9 France
benham@gia.univ-mrs.fr

Abstract

CLP(BNR) is a constraint system based on relational interval arithmetic and forms part of BNR Prolog /v.4. This is a much more powerful system than previous versions and allows a much wider class of problems to be handled, including discrete domain problems and boolean equations. It is also integrated more closely into Prolog, thus providing for smoother and more flexible interaction between Prolog and the constraint system. This paper describes the programming interface and gives some simple programming examples.

1 Introduction

The problems of providing a logical form of arithmetic for use in Prolog are well-known. The difficulties of doing correct computations with floating point are also well-known. One mechanism for overcoming both of these problems — applying a Prolog-like narrowing mechanism to intervals — was first suggested by Cleary [Cleary 1987]. These ideas were first fully implemented at Bell-Northern Research (BNR) in BNR Prolog in 1987 [BNR Prolog 1988] and have been successfully applied to problems far more complex than those described in [Cleary 1987]. Although similar in intent, Cleary's mechanism differs substantially from other constraint logic programming languages such as CLP($\Re$) [Jaffar and Michaylov 1987] and Prolog-III [Colmerauer 1990] in that it is not based on term-rewriting or symbolic equation solving techniques. In many respects the interval constraint system of BNR Prolog most closely resembles CHIP [Dincbas, Simonis and Van Hentenryck 1990] in so far as it is based primarily on local propagation techniques, and the system described by Hyvonen in [Hyvonen 89] which also handles continuous quantities.

A general description of the technique of relational interval arithmetic and its abstract semantics is given in [Older and Vellino 1993]. Briefly, the description of the problem given by the user is compiled into a constraint network whose nodes are instances of the primitive relations supported by the system. The variables of the problem are associated with a set (regarded as a state) determined by their upper and lower bounds; the states are partially ordered by set inclusion. The constraint network then defines an operator on states which is monotone and contracting with respect to the partial order on states, and which is also idempotent and correct. Correctness here means that no valid solution (e.g., in the theoretical real numbers) is ever eliminated during contraction; as a consequence of correctness the separation of multiple solutions must involve a mechanism such as backtracking or or-parallelism. This technique, since it works by removing non-solutions, can be regarded as a model elimination proof procedure, and this gives it quite different charateristics from constructivist exact arithmetic systems.

As shown in [Benhamou and Older 1992], a major advantage of this approach is that it provides a uniform treatment of a wide range of problems usually treated by very different methods. CLP(BNR) currently deals with boolean variables, the natural numbers, and the reals, and supports a large number of primitive relations. Because they share the same fundamental framework, it is possible to mix all three types of variables in a single problem. In the reals, it can deal with general non-linear functions, transcendental functions, and non-continuous functions on the same footing.

Because the underlying representation is based on intervals, this technique can also deal directly with the low-precision data and tolerance limits characteristic of most real-world applications. It is also for this reason an ideal tool for doing sensitivity analysis on complex models.

This paper is concerned solely with describing the user's view (and programming model) of CLP(BNR) by means of examples.

2 CLP(BNR)

CLP(BNR) is a sublanguage of BNR-Prolog for dealing with relations over the booleans, naturals, and reals. Because it is a separate sub-language, with its own notion of equality (==), it does not alter the usual syntactic meaning of Prolog unification (=). The two systems are closely coupled: both are relational, they share the notion of failure and backtracking, and bindings made by either system are visible in both. Prolog serves as a meta-language for CLP(BNR); in particular, since programming can be done in Prolog there is no need for programming constructs in CLP(BNR) proper.

The variables which are shared between the two systems are called intervals. An interval is a logic variable which has been constrained to only take numeric values (either integer or real) and is possibly further constrained to lie in a (closed) interval of the real line with (floating-point expressible) bounds. An interval can be created using syntax of the form:

```
V:real(LB,UB)   V:integer(LB,UB)   V:boolean
```

where `LB` and `UB` are either expressions which evaluate to numeric values or they are unbound variables, representing $\pm\infty$. Arithmetic relations on intervals, i.e., constraints, will either fail (in the usual Prolog sense), or will succeed, in which case the bounds of the intervals may have been changed (narrowed).

The following simple queries should give some feel for the CLP(BNR) system. (CLP(BNR) generally employs familiar syntax for arithmetic relations and functions, but details can be found in the Appendix.)

Establishing a constraint propagates information from the known to the less known

```
?- X:real(1,3),   Y**2==X.
   ==>  X : real(1.0, 3.0),
        Y : real(-1.73205080756888, 1.73205080756888)
```

Here is the same relation with both integer arguments; note that `X` becomes bound:

```
?- X:integer(1,3), Y:integer, Y**2==X.
   ==>  X : 1
        Y : integer(-1, 1)
```

If `Y` is constrained to be positive, then `Y` also becomes bound to its unique answer:

```
?- X:integer(1,3), Y:integer, Y**2==X, Y>0.
   ==>  X : 1
        Y : 1
```

Similar rules apply to general boolean relations:

```
?- B:boolean, 1 == B and (C or ~D) .
   ==>  B : 1
        C : boolean
        D : boolean

?- B:boolean,   1 == B and (C or ~D),   0 == B and C .
   ==>  B : 1
        C : 0
```

In some cases where an equation has a unique solution, equation solving is automatic:

```
?- [X,Y]:real, 1 == X + 2*Y,  Y - 3*X == 0.  % pair of linear eqns.
   ==>  X : real(0.142857142857143, 0.142857142857143)
        Y : real(0.428571428571429, 0.428571428571429)
```

Here is a more interesting example, but with non-linear (including transcendental) equation solving:

```
?- [X,Y]:real,X>=0,Y>=0,  tan(X) == Y, X**2+Y**2 == 5.
   ==>  X : real(1.09666812870547, 1.09666812870547)
        Y : real(1.94867108960995, 1.94867108960995)
```

Note that although the upper and lower bounds in these answers print the same at this printing precision, the internal binary forms must differ by at least one bit in the last place, or else the variables would have been bound to the exact answer.

For more complex problems, which may have multiple solutions, there is a "solve" predicate which separates the solutions (by backtracking) and forces convergence. For example, for polynomial root finding :

```
?- X:real(0,1), 0== 35*X**256 - 14*X**17 + X, solve(X).
   ==>  X : 0.0                                          % 1st sol.

   ==>  X : real(0.847943660827315, 0.847943660827315) % 2nd sol.

   ==>   X : real(0.995842494200498, 0.995842494200498) % last sol.
```

Similarly, for a pair of simultaneous non-linear equations :

```
?- [X,Y]:real,  X**3 + Y**3 ==2*X*Y, X**2+Y**2==1, X>=0, solve(X).
   ==>  X : real(0.391018886096038, 0.391085781049752)
   Y : real(-0.920382654506382, -0.92035423172858)

   ==>  X : real(0.449060394395367, 0.450226789190836)
        Y : real(0.892914239048135, 0.893501405810577)

   ==>  X : real(0.892906985142645, 0.893513338815017)
        Y : real(0.449036650353057, 0.450241175242194)
```

These examples have all been pure CLP(BNR) problems. The next few examples use a mixture of Prolog and CLP(BNR), although they can, if one chooses, equally well be regarded as programming in CLP(BNR).

3 Scheduling

The following example of a critical path algorithm computes a minimal completion time for a collection of activities with precedence relations[1]. First we create a "language" for describing precedence relations between activities:

```
op(700, xfx, before).                    % the precedes relation

activity(Name, Duration, task(Name,Start, Finish) ) :-
  [Start, Finish] : real,
        Finish == Start + Duration.

task(_,_,Finish) before task(_,Start,_):- Finish =< Start.
```

A particular scheduling problem can then be easily formulated, e.g.:

[1]Note this is scheduling over the Reals.

```
project( Start, Finish, [A,B,C,D,E,F,G]):-
  activity(a,10,A), Start before A,
        activity(b,20,B), Start before B,
        activity(c,30,C), Start before C,
        activity(d,18,D), A before D, B before D,
        activity(e, 8,E), B before E, C before E,
        activity(f,3,F), D before F,
        activity(g,4,G), F before G,
        G before Finish.
```

The actual computation of the shortest completion time and the start and finish windows for each task is then done by:

```
?- activity(start,0,S), activity(end,0,E), project(S,E,List),
   lower_bound(E).
```

3.1 Disjunctive Scheduling (Resource Limits)

Boolean constraints in BNR Prolog can be used not just for expressing pure boolean constraint problems (e.g. digital circuits), but are also useful in mixed computations, such as for disjunctive scheduling problems. For example, suppose that `S` is the start and `D` the duration of the task `T`. Now suppose that `T1` and `T2` (beginning at `S1`, `S2` and of duration `D1`, `D2` respectively), which are subject to precedence constraints, share the same resource and cannot execute concurrently (i.e. must be scheduled disjunctively). This constraint could be expressed by the formula:

```
(S1 + D1 =< B2) xor (B2 + D2 =< B1) == 1
```

where the **xor** constraint expresses the fact that the constraints are exclusive.

This way of expressing the problem provides an efficient alternative to the standard Prolog way of expressing disjunctive constraints by introducing a choicepoint. It can be applied effectively to solve scheduling problems with mutually exclusive resource allocation such as the bridge problem described in [Van Hentenryck 1989]. The CLP(BNR) program for this problem, which involves forty-six tasks and more than six hundred constraints, is described in detail in the appendix to [Benhamou and Older 1992].

4 Magic Series

Mixed boolean and integer constraints can be used to solve puzzles like the magic series [Colmerauer 1990, Van Hentenryck 1989]. The problem here is to find a sequence of non-negative integers $(x_0, ...x_{n-1})$ such that, for every i in $\{0, ..., n-1\}$, x_i is the number of occurences of the integer i in the sequence. In other words, for every $i \in \{0, ..., n-1\}$

$$x_i = \sum_{j=0}^{n-1} (x_j = i),$$

where the value of $(x = y)$ is 1 if $(x = y)$ is true and 0 if $(x \neq y)$ is true. Moreover, it can be shown that the two following properties are true.

$$\sum_{i=0}^{n-1} x_i = n, \sum_{i=0}^{n-1} i\,x_i = n$$

The BNR Prolog program that expresses these constraints is as follows:

```
magic(N,L) :-
  length(L,N),
  L : integral(0,_),
  constraints(L,L,0,N,N),
```

```
  firstfail(L).

constraints(L,[],N,0,0).
constraints(L,[X|Xs],I,S,S2) :-
  sum(L,I,X),
  J is I + 1,
  constraints(L,Xs,J,S1,S3),
  S==X + S1,
  S2 == X* I + S3.
  sum([],I,0).

sum([X|Xs],I,S) :-
  sum(Xs,I,S1),
  S== (X == I) + S1.
```

Thus the query that computes the magic series of length 4 produces the following results:

```
?- magic(4,L)
?- magic(4, [1, 2, 1, 0]). ;
?- magic(4, [2, 0, 2, 0]). ;
```

5 Non-Linear Continuous Constraints

This example is adapted from [Hong 1993] and serves to illustrate the application of interval arithmetic to continuous non-linear problems. (This case is equivalent to a pair of fourth degree polynomial inequalities.) This problem involves hitting a bird of known size and trajectory with a stone which has a fixed initial velocity and can only be launched at certain discrete instants. The problem is then to choose which instant and what elevation to use. The problem illustrates the logical nature of interval arithmetic, which makes it very natural in the context of Prolog.

The bird is modeled as a horizontal line segment of length `L`. The point `(X0,Y0)` is the bird initially and is located at `(0,H)`.

```
bird_init(X0,Y0,H,L):- X0>=0,X0=<L, Y==H.
```

The bird moves horizontally to the right with speed `U` and `(Dx,Dy)` is the displacement of the bird at time `T`.

```
bird_displacement(T,Dx,Dy,U):- Dx==U*T, Dy==0.
```

Then `(X,Y)`, the location of the bird at time `T`, is given by:

```
bird(T,X,Y,H,L,U) :-
  [Dx,Dy]:real,
   bird_displacement(T,Dx,Dy,U),
   bird_init(X-Dx, Y-Dy,H,L).
```

The projectile is modeled as a point; the (first) stone is initially located at `(0.0,0.0)`

```
stone_init(X,Y):- X==0.0,Y==0.0.
```

Note that the constants are not moved into the head of the clause; this will enable us to call `stone_init` with expressions for arguments later. (An expression, of course, will not generally unify with the constant `0.0`.)

Stones are shot with initial velocity `(Vx,Vy)`. The gravitational acceleration on earth is taken as `-9.8 m/sec*sec`, although an interval would be better. Let `(Dx,Dy)` be the displacement of the first stone at time `T`. Then the simplest formulation for the projectile motion is:

```
stone_displacement(T,Dx,Dy,Vx,Vy):-
  G is 9.8,
  Dx == Vx*T,
  Dy == Vy*T -  0.5*G*T**2 .
```

Then let (X,Y) be the position of a stone at time T:

```
stone(T,X,Y,Vx,Vy):-
  [Dx,Dy]:real,
  stone_displacement(T,Dx,Dy,Vx,Vy),
  stone_init(X-Dx,Y-Dy).
```

Stones are shot at time intervals of Dt; now let (X,Y) be the N-th stone at time T.

```
stone_N_th(T,X,Y,Vx,Vy,Dt,N):- stone(T-Dt*(N-1),X,Y,Vx,Vy).
```

The main procedure declares the basic variables and sets up the remaining constraints. The speed of shooting V is defined by V**2==Vx**2 + Vy**2; S is the slope of the initial stone trajectory and is restricted to be less than 1 (i.e. 45 degrees). Note that the last two calls equate the position of the bird and the position of the stone at the same time, i.e., the stone hits the bird.

```
hit(H,L,U,V,S,Dt, N, T,X,Y,Vx,Vy):-
  N:integer,            % shot number
  Dt:real,              % time between shots
  H:real,               % height of bird
  L:real,               % length of bird
  U:real,               % x-velocity of bird
  V:real,               % initial speed of stone
  S:real,               % angle of stone trajectory when fired
  T:real,               % time of hit
  [X,Y]:real,           % position of stone
  [Vx,Vy]:real,% initial velocity of stone
  V**2 == Vx**2 + Vy**2, % magnitude of velocity is speed
  Vx >=0, Vy>=0,% only fire up and to right
  Vy== S*Vx,            % slope restriction
  S >=0, S=<1.0,
  T>=0,                 % only consider positive times
  X>=0, Y>=0,           % and positions
  bird(T,X,Y,H,L,U),    % location of bird is same as
  stone_N_th(T,X,Y,Vx,Vy,Dt,N).   % location of stone
```

This initial formulation can now be run with specific data(bird 20 units high, second shot, etc.) :

```
?-hit(20.0,0.2,10.0,30.0,S,2.0,2,T,X,Y,Vx,Vy).
```

The answer (although a somewhat imprecise one) given by narrowing alone is then:

```
S : real(0.151327378188617, 1.0)
T : real(3.01569538493581, 7.4455056410137)
X : real(30.1518445255967, 75.9542677273962)
Y : real(20.0, 20.0)
Vx: real(5.53437068774039, 29.6622317367272)
Vy: real(4.59701024856432, 29.5097324297734)
```

The large size of these intervals can be reduced by several methods; the easiest method is to use solve on one of the variables:

```
?-hit(20.0,0.2,10.0,30.0,S,2.0,2,T,X,Y,Vx,Vy), solve(T).
```

This yields the much narrower answer:

```
S : real(0.878650132936296, 0.943989215521237)
T : real(3.63774689050188, 3.67373888991241)
X : real(36.3774689050188, 36.9373888991241)
Y : real(20.0, 20.0)
Vx: real(21.8153513340085, 22.5364861715658)
Vy: real(19.8016865705633, 20.5934563921109)
```

The size of the intervals is now determined mainly by the size of the bird. If the bird length is reduced to a point, these intervals also become near points, with the residual intervals reflecting round-off errors.

Another approach which is sometimes very useful is to add a redundant constraint

```
0 =< Vy - G*T
```

to the predicate **stone_displacement**. Although this merely expresses the fact that the stones's trajectory is everywhere lower than its highest point, even without solve it yields the answer:

```
S : real(0.878438127193421, 1.0)
T : real(3.59504189240291, 4.06175182228815)
X : real(35.9504189240291, 40.8175362967534)
Y : real(20.0, 20.0)
Vx: real(19.7989898732233, 22.5388553391693)
Vy: real(19.7989898732233, 22.5388553391693)
```

Finally, if we replace the formula

```
Dy== Vy*T - 0.5*G*T**2$$
```

in **stone_displacement** by the mathematically equivalent expression (formed by "completing the square")

```
Dy== (Vy*Vy)/(2*G) - (G/2)*(T - Vy/G)**2
```

we get — without solve — the answer:

```
S : real(0.895608876389984, 0.924134061383734)
T : real(3.60762768765977, 3.7015123584053)
X : real(36.0436531011464, 37.2740024982711)
Y : real(20.0, 20.0)
Vx: real(22.0296057795281, 22.352752887285)
Vy: real(20.0297702035754, 20.3540924671305)
```

Other questions, such as finding the solution(s) when the number of the stone is unknown, can also be easily formulated, e.g.:

```
?- N:integer(1,10), hit(20.0,0.2,10.0,30.0,S,2.0,N,T,X,Y,Vx,Vy),
   enumerate([N]).
```

Note that this problem involves both continuous and discrete variables.

6 Concluding Remarks

In problems involving pure integer and pure boolean constraints we have been able to compare the performance of the initial version of CLP(BNR) with other constraint systems specialized for these domains. The performance of CLP(BNR) appears to be roughly comparable to other systems (within a small linear factor) in most cases so far examined. The major exception is problems consisting mainly of not-equal relations, for which interval representations are not well suited.

One of the advantages of this approach is that the uniform semantics makes it easy to handle cases involving a mixture of integers, booleans, and reals in the same problem. We have found that in many cases, such as disjunctive scheduling as discussed above, the use of mixed type interval formulations leads to significant performance improvements over previously available techniques, yet also simplifies the problem formulation.

One area of our ongoing research is that of continuous, non-linear constrained optimization problems, where this technology looks very promising. In marked contrast to conventional approaches, a useable general algorithm for this class of problems in BNR Prolog is a succinct direct encoding of the classical mathematical theory of such problems, and its execution provides in principle a formal proof of optimality as well as a numerical solution.

The application of this technology to really complex problems is not always straightforward, because different formulations of the same problem can sometimes have quite different performance characteristics. Finding "good" formulations is therefore still a process of discovery, guided by experience, intuition, and a handful of basic principles, such as the deferring of choices, the first fail principle, and the controlled use of redundancy. Once found, however, such formulations are usually transparently clear (but possibly subtle) statements of the problem.

Acknowledgements

The authors wish to acknowledge the contributions of R. Workman, J. Rummell, and A. Vellino of Bell Northern Research for their myriad contributions to this project.

References

[Benhamou and Older 1992] Benhamou F. and Older W. "Applying Interval Arithmetic to Real, Integer and Boolean Constraints" BNR Research Report, 1992.

[BNR Prolog 1988] BNR Prolog User Guide and Reference Manual, BNR 1988.

[Cleary 1987] Cleary, J. C. "Logical Arithmetic", Future Computing Systems. 2 (2). pp.125-149, 1987.

[Colmerauer 1990] Colmerauer, A. "An Introduction to Prolog-III", Communications of the ACM, July 1990 p.70-90.

[Dincbas, Simonis and Van Hentenryck 1990] Dincbas M., Simonis H. and van Hentenryck P. "Solving Large Combinatorial Problems in Logic Programming", Journal of Logic Programming 8, 1&2, pp. 72-93, 1990.

[Hyvonen 89] Hyvonen, E. (1989) "Constraint Reasoning Based on Interval Arithmetic", Procedings of IJ-CAI 1989 pp. 193-199.

[Jaffar and Michaylov 1987] Jaffar, J. and Michaylov, S. "Methodology and Implementation of a CLP system", Proc. 4th Int. Conf. on Logic Programming, J- L. Lassez (Ed), MIT Press, 1987.

[Hong 1993] Hoon Hong, "RISC-CLP(Real):Logic programming with Non- linear constraints over the Reals" in Constraint Logic Programming: Selected Research. F. Benhamou and A. Colmerauer, eds. MIT Press, 1993, to appear.

[Older and Vellino 1993] Older, W., "Constraint Arithmetic on Real Intervals" in Constraint Logic Programming: Selected Research. F. Benhamou and A. Colmerauer, eds. MIT Press. 1993, to appear.

[Older and Vellino 1990] Older, W. and Veliino,AJ., "Extending Prolog with Constraint Arithmetic on Real Intervals" , Proceedings of the Canadian Conference on Electrical and Computer Engineering, 1990

[Van Hentenryck 1989] Van Hentenryck P. Constraint Satisfaction in Logic Programming, MIT Press, Cambridge, 1989.

Appendix

Table 1: Declarations and Definitions

Syntax	Name	Description
`X: Type(LB,UB)`	declaration	constrains logic variable(s) `X` to be of specified type and range
`X is Y`	definition	variable `X` takes the type of expression `Y`
`X := Y`	definition	same as `is` but only does interval arithmetic

Table 2: First-Order Relations

Syntax	Name	Description
`X == Y`	arithmetic equality	`X` and `Y` constrained to be equal
`X =< Y`	less than or equal	`X` is constrained to be less than or equal to`Y`
`X >= Y`	greater than or equal	`X` is constrained to be greater than or equal to `Y`
`X < Y`	strict less than	(may be unsound on real intervals)
`X > Y`	strict greater than	(may be unsound on real intervals)
`X <> Y`	dif, inequality	`X` and `Y` (both integer) are constrained to be distinct

Table 3: Second-Order Relations

Syntax	Name	Description
`X <= Y`	inclusion	`X` is constrained to be a subinterval of `Y`
`X |= Y`	start together	`X` and `Y` are constrained to have the same lower bound
`X =| Y`	end together	`X` and `Y` are constrained to have the same upper bound

Table 4: Dyadic

Operation	Type Signature	Restrictions
`Z := X + Y`	`I := I + I` or `R :=R + R`	
`Z := X - Y`	`I := I - I` or `R :=R - R`	
`Z := X * Y`	`I := I * I` or `R :=R * R`	
`Z := X / Y`	`R:= I / I` or `R:=R / R`	`Y=0, X<>0` automatically excluded
`Z := min(X,Y)`	`I:= min(I , I)` or `R :=min(R , R)`	
`Z := max(X,Y)`	`I:= max(I, I)` or `R :=max(R , R)`	
`Z := (X ; Y)`	`I := (I ; I)` or `R :=(R ;R)`	(means `Z==X or Z==Y`)

Table 5: Monadic

Operation	Type Signature	Restrictions
Z := X ** N	I := I ** I ; R :=R ** I,	N must be an integer constant
Z := -X	I := -I ; R := -R	
Z := abs(X)	I := abs(I) ; R := abs(R)	Z >= 0
Z := sqrt(X)	R := sqrt(R)	Z>=0, X>=0
Z := exp(X)	R := exp(R)	Z>0
Z := ln(X)	R := ln(R)	X>0
Z := sin(X)	R := sin(R)	-1=<Z=<1
Z := asin(X)	R := asin(R)	-1=<X=<1,-pi/2=<Z=<pi/2
Z := cos(X)	R := cos(R)	-1=<Z=<1
Z := acos(X)	R := acos(R)	-1=<X=<1, -pi/2=<Z=<pi/2
Z:= tan(X)	R := tan(R)	-1=<Z=<1
Z := atan(X)	R := atan(R)	-1=<X=<1, -pi/2=<Z=<pi/2

Table 6: Boolean

Operation	Type Signature
Z := X and Y	B:= B and B (and)
Z := X nand Y	B:= B nand B
Z := X or Y	B:= B or B (inclusive or)
Z := X nor Y	B:= B nor B
Z := X xor Y	B:= B xor B (exclusive or)
Z := X	B:= B (boolean negation)

Table 7: Mixed

Operation	Type Signature	Restrictions
Z := (X==Y)	B:=(R== R)	test/impose equality
Z := (X<>Y)	B:=(R<> R)	equiv. Z := (X==Y)
Z := (X=<Y)	B:= (R=<R)	test/impose inequality
Z := (X>Y)	B := (R>R)	equiv. Z := (X=<Y)
Z := (X>=Y)	B := (R>=R)	equiv. Z := (Y=<X)
Z := (X<Y)	B:=(R< R)	equiv. Z := (X>Y)

Table 8: Miscellaneous

Syntax	Semantics
lower_bound(X)	interval X is narrowed to it lower bound
upper_bound(X)	interval X is narrowed to it upper bound

Table 9: Pseudo-functions on intervals

Syntax	Result
midpoint(X)	returns midpoint of an interval, i.e. (UB-LB)/2
median(X)	returns a point in an interval (suitable for splitting)
delta(X)	returns the width of an interval (i.e. UB - LB)

Table 10: Control and Enumeration Predicates

Predicate	Description
`interval(X)`	true if variable `X` is an interval, fails otherwise.
`domain(X, Type(L,U))`	get the type and bounds of interval `X`, fails otherwise.
`range( X, [L,U])`	queries the lower and upper bounds of an interval or numeric
`enumerate(IntervalList)`	systematic enumeration of a list of intervals (usually used on integer and boolean intervals).
`firstfail(IntegerIntervalList)`	systematic enumeration of a list of integer intervals *in dynamic order of size*.
`solve(X, N, Eps)`	a nondeterministic subdivision algorithm for real intervals which stops after `N` levels or when `delta(X)` becomes less than `Eps*(initial delta(X))`. `X` may also be a list of intervals.
`solve(X)`	the same as `solve(X,6,0.0001)`. Produces a maximum of 2**6 answers) *(Note 3)*
`presolve(X)`	an intelligent solve for use with complex problems with many variables
`absolve(X, N)`	deterministic predicate to trim the ends of interval `X` to make it as small as possible. `N` is a small integer (¡ 32) which controls the relative precision. `X` may be a list of intervals.
`absolve(X)`	same as `absolve(X,14)`
`degrees_of_freedom(X,E,R,I,D)`	returns the number of equations `E`, continuous variables `R`, inequalities `I`, and discrete (integer or boolean) variables `D` in the constraint network attached to interval `X`.
`constraint_network(X, N where S)`	returns a description of the network attached to interval `X` and its current state.

Robot Programming and Constraints *

Dinesh K. Pai
Computer Science Department
University of British Columbia
pai@cs.ubc.ca

Abstract

Constraints play a central role in the analysis and planning of robot motion. We suggest that constraints form an appropriate language with which to program robot motion as well and describe the Least Constraint framework.

1 Introduction

Robot programming has largely been based on specifying motions in terms of trajectories in a configuration space or a state space. For example, motions of industrial robot manipulators are usually specified in terms of the trajectory of the hand.

Such specifications are simple and intuitive and adequate for some applications. But their deficiencies are increasingly apparent. For example, consider the problem of programming a human-like robot to walk dynamically in three dimensions [Pai90]. The robot has a large number joints which have to be used to achieve a desired motion of the body. One approach to programming such a task is to pick some periodic trajectory for the joints, and attempt to track it. However, it is not clear that this is the natural characterization of the task.

Some important issues:

- How to specify trajectories for high degree of freedom (also called "redundant") robots? Such robots are increasingly common since they improve versatility and fault tolerance.

  Most natural specifications for these robots will only partially constrain the motion. There has been considerable research into "redundancy resolution", i.e., the generation of unique trajectories from partial specifications by the introduction of other criteria such as singularity avoidance (e.g., [HS85]). But the problem of "redundancy maintenance," i.e., accepting and maintaining partial specifications remains.

- How should a trajectory specification deal with objects in the robot's environment? It was quickly realized that in order to perform interesting tasks, a robot has to interact with its environment — to avoid some objects (obstacles) [LP81] and to manipulate others [MS85]. A motion specification should account for the geometric constraints imposed by the objects. This problem has usually been solved by using a separate motion planner which generates a safe trajectory for a low-level trajectory executor. However since the motion constraints are not known at run-time, the executor cannot safely alter the planned trajectory in response to sensed events.

- How to combine motions at run-time? The motion planning task is often divided among different modules whose outputs are combined to produce the actual motion. Since trajectory specifications are entirely procedural most researchers have used mutual exclusion to arbitrate among the modules, leading to run-time behavior that is difficult to anticipate.

*Supported in part by NSERC and the Institute for Robotics and Intelligent Systems. Support for this work at Cornell University was provided in part by ONR Grant N00014-88K-0591, ONR Grant N00014-89J-1946

Returning to our example of the human-like walking machine, we would instead like to program such machines incrementally, by specifying assertions about its behavior. We can specify several requirements for walking: for instance, (i) the foot should clear the ground during the swing phase of the leg, (ii) the swing foot should be moved to a location suitable for dynamic balance by foot placement, and so on. The machine is controlled to satisfy these requirements at run-time.

It may turn out that the initial requirements were inadequate — for example, one may find that there is nothing to prevent knee flexion from becoming so large that walking is impossible. In this case one would like to modify the existing program by merely adding new assertions: for example, by adding the assertion that the pelvis should be above a certain height. This is not possible in current robot programming languages.

The Least Constraint (LC) framework was designed to address these problems [Pai89, Pai91]. It has been successfully used to program dynamic walking in a simulated human-like biped and will be used to program a new class of high degree of freedom robots that are under construction in our lab.

2 Constraints in Robotics

2.1 Robot domains: configuration space, task space

The behavior of robots is usually described in two types of spaces:

A *configuration space* of the robot is a space whose points uniquely describe all possible positions and orientations of every part of the robot. It is usually understood that the dimension the configuration space is the minimum required. Otherwise, it can be called a *descriptor space*.

The desired behavior, on the other hand, may be more conveniently expressed in other spaces called *task spaces*. For example, the position and orientation of the gripper of a robot is a task space used by most industrial robot programming languages.

The configuration and task spaces are usually differentiable manifolds, but not necessarily covered by a single chart. The space of rigid body rotations, $SO(3)$, is a common example.

Typical spaces include:

- Joint space: the configuration space of a chain of rigid bodies with one degree of freedom joints. Most industrial robots are of this type. A robot with r revolute joints and p prismatic joints has a joint space $\equiv T^r \times \mathbb{R}^p$.
- The space of rigid motions: In three dimensions, this is the manifold of the Special Euclidean group, $SE(3) \equiv \mathbb{R}^3 \times SO(3)$. In two dimensions, this is $SE(2) \equiv \mathbb{R}^2 \times \mathrm{S}^1$. The desired configuration of an object manipulated by the robot or of a link of the robot could be expressed in this space.

2.2 Geometric Constraints

Interaction between a robot and solid objects in its environment impose a natural set of constraints on its motion and have been extensively studied in motion planning (see e.g., [Yap87, Lat91]). The solid objects are usually modeled in world coordinates and can be transformed into configuration space (see [LP83, Don87, Lat91]).

For example, consider the configuration space of a mobile robot modeled as a rectangle free to move in the plane; its configuration space $\mathcal{C}$ is $\mathbb{R}^2 \times S^1$ with coordinates $\mathbf{x}$ for $\mathbb{R}^2$ and θ for S^1. A configuration space obstacle CO_{jk} due to a convex subset $\mathcal{A}_j$ of the robot interacting with a convex subset $\mathcal{B}_k$ of the obstacle can be defined by a family of constraints [LP83, Don88],

$$C_i^{jk} := (\theta \in A_i) \rightarrow (f_i(\mathbf{x}, \theta) \leq 0)\,. \tag{1}$$

Here $f_i : \mathcal{C} \mapsto \mathbb{R}$ with positive f_i corresponding to free space. A_i is the applicability set, which in this case is of the form

$$(\theta \leq \theta_{max,i}) \bigwedge (\theta_{min,i} \leq \theta). \tag{2}$$

The configuration space obstacle is given by

$$\bigwedge_{i \in family(\mathcal{A}_j, \mathcal{B}_k)} C_i^{jk}. \tag{3}$$

In the general case when the robot $\mathcal{A}$ and obstacles $\mathcal{B}$ are non-convex, they can be decomposed into (possibly overlapping) convex polygons $\mathcal{A}^j$ and $\mathcal{B}^k$, so the general form of the free configuration space around the obstacle is

$$\neg\left(\bigvee_{j,k}\bigwedge_{i\in cfamily(\mathcal{A}_j,\mathcal{B}_k)} C_i^{jk}\right). \tag{4}$$

2.3 Motion constraints

The physics of the robot and its interaction with the world imposes additional constraints on its motion. The exact constraints depend on our model of the robot and its world. Modeling remains a task involving engineering judgement regarding the significant physics. It is important to note that unlike the geometric constraints above these are *differential-algebraic* constraints.

The following are constraints resulting from some commonly useful models.

2.3.1 Differential kinematic constraints

Simple examples of such constraints are bounds due to actuator limits or safety considerations of the form

$$\dot{q}_{min} \leq \dot{q} \leq \dot{q}_{max} \tag{5}$$
$$\ddot{q}_{min} \leq \ddot{q} \leq \ddot{q}_{max} \tag{6}$$

More interesting constraints arise from motion involving rolling without slipping, such as in wheels and fingers. In the above example of the mobile robot with $\mathbf{x} = (x, y)^T$, the constraint that the instantaneous motion of the center of a wheel is restricted to the plane of the wheel leads to a differential kinematic constraint

$$\dot{x}\sin\theta - \dot{y}\cos\theta = 0. \tag{7}$$

This constraint can be shown to be *non-holonomic*, i.e., it can not be integrated to produce a geometric constraint involving x, y, and θ *only*.

2.3.2 Dynamic constraints

The dynamics of the rigid bodies in the robot and the dynamics of the actuators can be significant for high performance robots. These lead to a system of highly non-linear, coupled differential equations which impose additional constraints. The formulation of these constraints for control and simulation has been extensively studied (see e.g. [RS88]). These can be broadly classified as

- State space constraints: the equations of motion are formulated in terms of a minimal set of Lagrangian coordinates $\mathbf{q}$, leading to a system of ordinary differential equations. For example if a robot is constructed as a chain of rigid bodies, the equations of motion are of the form

$$\dot{\mathbf{q}} = \mathbf{v}, \tag{8}$$
$$M(\mathbf{q})\dot{\mathbf{v}} = \tau - f(\mathbf{q}, \mathbf{v}), \tag{9}$$

  where M is a symmetric positive-definite inertia tensor and τ_i is the generalized force applied along q_i.

- Descriptor space constraints: the equations of motion are formulated in terms of a sufficient, but not necessarily minimal, set of coordinates, $\mathbf{x}$. Additional constraints are imposed between the coordinates, leading to differential-algebraic equations of the form

$$\dot{\mathbf{x}} = \mathbf{v}, \tag{10}$$
$$M(\mathbf{x})\dot{\mathbf{v}} = f(\mathbf{x}, \mathbf{v}) + G^T(\mathbf{x})\lambda \tag{11}$$
$$0 = g(\mathbf{x}), \tag{12}$$

  where M is a (different) symmetric positive-definite inertia tensor, $G = [Dg]$, and λ are Lagrange multipliers. A common example is to take the position and orientation of each rigid body as descriptors,

and introduce constraints for each joint of the robot. It is easier to formulate the dynamics of closed kinematic chains using this approach. However, equations in descriptor form can be difficult to integrate for high index problems [BCL89, AP91].

3 Least Constraint (LC)

3.1 Program constraints

While the system constraints above have been used in the analysis and planning of robot motion, user specification of desired robot behavior has been in terms of either a trajectory of the robot or a goal state [1]. In the former case the motion is explicitly specified and the main focus has been on the control of the robot to track this trajectory. In the latter case, the motion of the robot is implicitly specified and the focus has been on motion planning, i.e. the construction of a trajectory which reaches the goal.

We believe that specifying robot motions with constraints is appropriate and natural in many situations. A simple example is toleranced motion, in which we want to follow a trajectory but within a generous tolerance of the nominal trajectory. Most robot motions are of this type. Another example is the idea of a "funnel" [Mas85]. Here we only require that the robot lie in a set that contracts over time until a desired set of configurations is attained.

3.2 LC

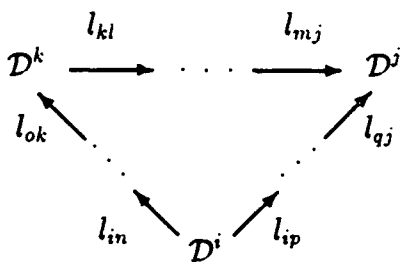
In the LC framework, motions are expressed by means of time- and state-dependent assertions [Pai91]. These assertions are defined using inequality constraints which describe the set of allowed states as a function of time. The constraints are solved at run time to produce a satisfying motion.

Since complex mechanical systems have large state spaces, it is not convenient or natural to express all of the constraints in a single space. For instance, even the simple walking machine simulated in [Pai90] has a 28-dimensional state space. For convenience of expression, users define derived variables in terms of the basic (e.g., state) variables — an example of this is the definition of task and end-effector coordinates for robot manipulators. LC generalizes such constructions to allow the creation of arbitrary, user definable quantities which are natural to the tasks and the constraints being expressed. One can isolate small groups of variables into domains on which to focus. For example, the foot collision constraints in the above walking example are best expressed in a separate foot position domain.

In LC, users define a *domain system*, $\{\mathcal{D}^i : i \in I\}$, related by *linking functions*

$$l_{ij} : \mathcal{D}^i \to \mathcal{D}^j, \quad (i,j) \in L \subset I \times I, \tag{13}$$

which satisfy the basic consistency condition that all diagrams of the following form commute.



All domains $\mathcal{D}^i$ are connected to a *basic domain* $\mathcal{D}^0$ by compositions of linking functions :

$$\mathcal{D}^0 \xrightarrow{l_{0j}} \ldots \xrightarrow{l_{ki}} \mathcal{D}^i.$$

Briefly, the motivation for using domain systems is that they allow a constraint on a subdomain $\mathcal{D}^i$ to be lifted to an equivalent constraint on the basic domain $\mathcal{D}^0$ using compositions of linking functions. $\mathcal{D}^0$ is usually the configuration space or state space.

[1] A notable exception is the potential field approach to obstacle avoidance [Kha86] which shares many features with LC. LC generalizes obstacles to constraints in arbitrary domains. See [Pai91] for a discussion.

A ***motion specification*** in LC consists of a system of time-varying inequality constraints $C_\alpha, \alpha \in A$, on the domains $\mathcal{D}^i$; here each constraint C_α is expressed by

$$C_\alpha := f_\alpha(\mathbf{x}^i, \dot{\mathbf{x}}^i, t) \leq 0,$$

where $f_\alpha : \mathcal{D}^i \times T\mathcal{D}^i \times \mathbb{R} \rightarrow \mathbb{R}$ is a smooth map and $\mathbf{x}^i(t)$ denotes a time-dependent trajectory in $\mathcal{D}^i$. Such C_α and their conjunctions

$$C := \bigwedge_\alpha C_\alpha$$

are executable LC motion programs. The meaning of the constraint is that the robot is controlled to make the specified expressions $C_\alpha := f_\alpha(\mathbf{x}, \dot{\mathbf{x}}, t) \leq 0$ true at all times t.

3.3 Constraint Satisfaction

LC programs are executed by satisfying constraints at run-time. This produces a trajectory $\mathbf{x}(t) \in \mathcal{D}^0$ such that the derived constraints

$$\bar{C}_\alpha := f_\alpha((l_{ki} \circ \ldots \circ l_{0j})(\mathbf{x}(t)), t) \leq 0$$

obtained by lifting the original constraints using the linking maps are satisfied at all times t. In LC, this is done at discrete time steps $t_{(n)}$: at every time step $t_{(n)}$, a feasible point $x(t_{(n)})$ is produced, and is used to compute the control u.

Criteria for good constraint satisfaction algorithms for LC constraints are quite different than those encountered in other domains such as finite constraint satisfaction and large scale optimization. We first describe features of a suitable constraint satisfaction algorithm and then describe the constraint satisfaction algorithm we currently use.

- On-line constraint satisfaction. This is required since many of the constraints will only be known at run time.
- Any time solution [DB88]. The constraint satisfaction should be interruptible – if a feasible solution cannot be found in the allotted time an improved estimate should be returned.
- Exploit continuity. The constraints typically vary slowly relative to the rate at which they are satisfied. Since a similar problem was solved at the previous time step, a good starting guess is available.
- Exploit "large" feasible sets due to inequality constraints.

LC separates the specification of constraints and the techniques used for satisfying them. Different constraint satisfaction algorithms can be used. We first discretize the constraint function in time using numerical methods with good stability properties such as linear multistep methods or implicit Runge-Kutta methods [BCL89]. The step size $h > 0$ is taken to be the servo rate. For example, using the simple implicit backward-Euler scheme, the constraint

$$\dot{x} - \cos(\theta)v \leq 0$$

is discretized as

$$x_{(n)} - x_{(n-1)} - h\cos(\theta_{(n)})v_{(n)} \leq 0.$$

The discretization yields a set of algebraic constraints in $x_{(n)}$ at time $t_{(n)}$.

Next, the inequalities are solved to produce a solution at $t_{(n)}$. The solution is usually advanced a few time steps beyond the present time to produce a trajectory segment, combining simulation with trajectory generation. We currently use a conjugate direction method with a quadratic penalty function. We have also experimented with other methods such as relaxation methods [Pai91, Zha] and barrier methods. These methods are local and iterative. The locality is a limitation but makes the methods fast enough to be implementable in real time. The methods exploit knowledge of a good starting point for iteration. Since the feasible sets specified with inequality constraints are usually of non-zero measure and typically large, overrelaxation is used to speed convergence. The gradients are computed cheaply using automatic differentiation [PS93], which enables us to compute each update within only a small factor of the time to evaluate the constraint functions.

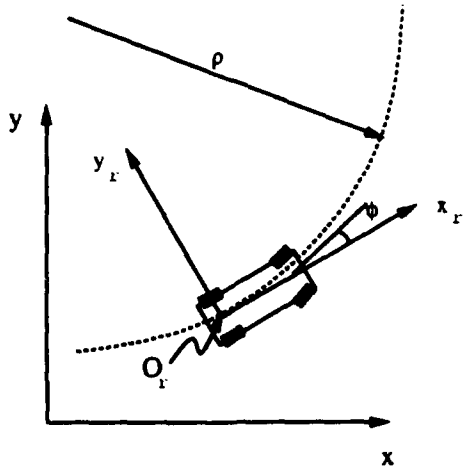


Figure 1: Mobile Robot Notation

Finally, the satisfying trajectory segment is sent to a low-level tracking controller which drives the actuators.

The method is fast and effective in practice. We should point out that the numerical solution of non-linear systems is a delicate matter. There are several practical considerations such as pre-scaling of constraints, maintaining active constraints and selecting numerical tolerances that are important for implementing such constraint solvers.

4 Example: Mobile robots

Here we demonstrate the LC approach in programming the motion of a mobile robot. This work is part of the Dynamo mobile robotics project at UBC led by Mackworth and Pai. The Dynamo facility includes the Dynamite testbed consisting of radio controlled cars whose absolute position is sensed at video rates using off-board vision.

While mobile robots are relatively low degree of freedom machines, a collection of mobile robots may be viewed as a single high degree of freedom robot. The presence of non-holonomic constraints also adds to the difficulty of specifying the motion of a wheeled robot. These constraints do not lower the dimension of configuration space, but on the other hand, the user cannot specify an arbitrary trajectory to track. By specifying the desired motion weakly in terms of the constraints on the required motion, the constraints can be solved at run-time to produce a motion that satisfies the constraint.

The dynamite mobile robots are front-wheel steered with a steering angle ϕ and can be controlled by setting a throttle value and steering angle. We have implemented a low-level PI controller which servos the robot to track a specified speed v_d and turning radius ρ_d. To a first approximation, the robot can be modeled as a kinematic machine, with two inputs v_d and ρ_d. Experiments with programming these mobile robots with LC are in progress, but we describe a simple example here.

Following the notation in [Lat91] (see Figure 4), let the configuration space C of the mobile robot be the space of displacements of a coordinate frame (O_r, x_r, y_r) attached to robot. C has coordinates x, y, θ. For simplicity, we take $\theta \in \mathbb{R}$ so C is a covering space of $SE(2)$. Define the forward speed $v = \dot{x}\cos(\theta) + \dot{y}\sin(\theta)$ and the turning radius $\rho = v/\dot{\theta}$. Note that ρ is a signed quantity, with $\rho > 0$ corresponding to the center of rotation of the robot lying to the "left" of the robot, i.e., having $y_r > 0$. However, since $\rho = \infty$ when the robot is moving along a straight line, we use the less natural but more convenient quantity $\varsigma = 1/\rho$.

The robot has the following constraints

Motion constraints

1. Rolling constraint:

$$C_1 := \dot{x}\sin(\theta) - \dot{y}\cos(\theta) = 0 \tag{14}$$

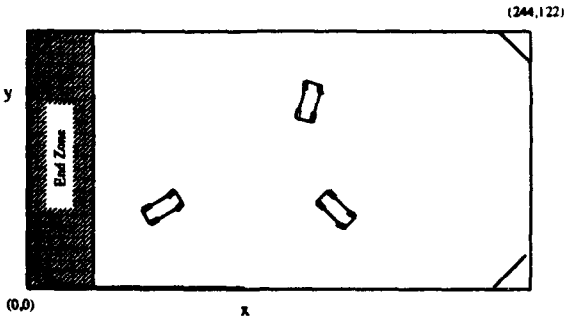


Figure 2: Dynamite robot pen

2. Speed limit:

$$C_2 := (v_{min} \leq v) \bigwedge (v \leq v_{max}) \tag{15}$$

3. Steering limit:

$$C_3 := (\varsigma_{min} \leq \varsigma) \bigwedge (\varsigma \leq \varsigma_{max}) \tag{16}$$

Geometric constraints

4. Wall and obstacle constraints: The robot is in a convex, enclosed pen populated by other robots (see figure 2). The pen and the robots are well represented by convex polygons. Details of representing the free space of the robot among such obstacles can be found in [LP83, Don88, Can86]. Here only type-B contact between an edge of the pen and a vertex of the robot needs to be considered and each contact mode generates a constraint in the configuration space. Let an edge w be at a distance d_w along the unit normal to the edge $(n_{w,x}, n_{w,y})$ oriented in the direction of free space. Contact between the edge and the vertex u of the robot whose coordinates are $(p_{u,x}, p_{u,y})$ in the (O_r, x_r, y_r) frame generates the configuration space obstacle constraint (see Equation 1)

$$C_u^w := (\theta \in A_{u,w}) \rightarrow ((x + p_{u,x}\cos\theta - p_{u,y}\sin\theta)n_{w,x}) + ((y + p_{u,x}\sin\theta + p_{u,y}\cos\theta)n_{w,y}) - d_w \leq 0 \tag{17}$$

User constraint

5. Pusher: The task is to reach the "end zone" $0 \leq x \leq 25$ at the left wall (see figure 2). This can be achieved by the constraint

$$C_p := x - x_{w,max} + st \leq 0 \tag{18}$$

where s is the speed of the pusher and $x_{w,max}$ is the x-coordinate of the right wall.

With more complicated obstacles, a motion planner can be used to process the constraints into a form suitable for LC. For example, Brooks [Bro83] describes a method for extracting a net of "freeways" which are generalized cylinders covering the free space of the robot. However instead of traveling along the spines of the cylinders as in [Bro83] we can use the freeways as constraints. The robot is pushed through a freeway by pusher constraint. The advantage of this approach is that additional constraints such as those due to unforeseen obstacles can be added run time.

5 Future Work

We are developing a new version of LC with a better software architecture, improved performance, and with additional capability to handle differential-algebraic constraints. This is required for new applications such as programming modular legged robots. The new implementation of LC in progress is designed to be efficient enough to run on a network of small embedded controllers.

In the long term, we are interested in the problem of verifying the correctness of LC programs for appropriate notions of correctness. This involves exact or approximate methods for detecting topological changes to the feasible set of system states and for estimating whether a connected component of the feasible set will collapse.

References

[AP91] U. Ascher and L. R. Petzold. Stability of computational methods for constrained dynamics systems. Technical Report, Computer Science Department, University of British Columbia, May 1991.

[BCL89] K. Brenan, S. Campbell, and L.Petzold. *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations.* North Holland, 1989.

[Bro83] R. A. Brooks. Solving the find-path problem by good representation of free space. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(3):190–196, 1983.

[Can86] John Canny. Collision detection for moving polyhedra. *IEEE Trans. PAMI*, 8(2), March 1986.

[DB88] T. Dean and M. Boddy. An analysis of time dependent planning. In *AAAI-88*, 1988.

[Don87] B. R. Donald. A search algorithm for motion planning with six degrees of freedom. *Artificial Intelligence*, 31(3), 1987.

[Don88] B. R. Donald. A geometric approach to error detection and recovery for robot motion planning with uncertainty. *Artificial Intelligence*, 37((1-3)):223–271, Dec. 1988.

[HS85] J. M. Hollerbach and K. C. Suh. Redundancy resolution of manipulators through torque optimization. In *IEEE International Conference on Robotics and Automation*, pages 1016 – 1021, 1985.

[Kha86] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90 – 98, 1986.

[Lat91] J. C. Latombe. *Robot Motion Planning.* Kluwer, 1991.

[LP81] Tomás Lozano-Pérez. Automatic planning of manipulator transfer movements. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(10):681–689, 1981.

[LP83] Tomás Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, February 1983.

[Mas85] Matthew T. Mason. The mechanics of manipulation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 544–548, 1985.

[MS85] Matthew T. Mason and J. Kenneth Salisbury, Jr. *Robot Hands and the Mechanics of Manipulation.* M.I.T. Press, 1985.

[Pai89] Dinesh K. Pai. Programming parallel distributed control of complex systems. In *Proceedings of the IEEE International Symposium on Intelligent Control*, pages 426–432, September 1989.

[Pai90] Dinesh K. Pai. Programming anthropoid walking: Control and simulation. Cornell Computer Science Tech Report TR 90-1178, 1990.

[Pai91] Dinesh K. Pai. Least constraint: A framework for the control of complex mechanical systems. In *Proceedings of the American Control Conference*, pages 1615 – 1621, 1991.

[PS93] D. K. Pai and T. H. S. Ser. Simultaneous computation of robot kinematics and differential kinematics with automatic differentiation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems '93*, 1993.

[RS88] R. E. Roberson and R. Schwertassek. *Dynamics of Multibody Systems.* Springer-Verlag, 1988.

[Yap87] C. Yap. Algorithmic motion planning. in *Advances in Robotics: Volume 1*, edited by J. Schwartz and C. Yap, Lawrence Erlbaum Associates, 1987.

[Zha] Y. Zhang. Constraint methods for specification and design of hierarchical control systems. Term Project for CPSC 532 -Computational Robotics, University of British Columbia, Spring 1992.

Constraints in Nonmonotonic Reasoning*

William C. Rounds and Guo-Qiang Zhang
Artificial Intelligence Laboratory
University of Michigan
Ann Arbor, MI 48109
(rounds, gqz)@engin.umich.edu

Abstract

We report on the use of constraints to govern partial models of first-order logic. These models are constructed using systems of default rules, as in the default logic of Reiter, but where Reiter's rules build logical theories, our rules build models. Our approach takes advantage of domain-theoretic notions. A system of default rules is a straightforward generalization of Scott's notion of information system, already an important tool in understanding constraint programming.

We apply our theory to resolve an anomaly due to Poole on the meaning of defaults. Using finite default models, constraints, and a constructive, rule-based notion of conditional degree of belief, we give a probabilistic way of interpreting default statements.

1 Introduction

This paper is concerned with an application of constraint theory, information systems, and domain theory in artificial intelligence. Specifically, we are interested in finding a proper semantic setting for Reiter's original work on default logic [10]. We would like to report on a new solution to this problem, which we call *default model theory*. This theory comes in several flavors. In this paper, we show how to give default model semantics to first-order logic. Doing this crucially involves the notion of *constraint* as a kind of law that governs the behavior of partial models of first-order logic.

Etherington [4] gave the first model-theoretic semantics to Reiter's logic. This was a system based on *sets* of first-order models. Marek, Nerode, and Remmel [8], gave a semantics for nonmonotonic rule systems. They translated Reiter's default rules into finitary formulas of a certain special infinitary logic. Extensions – the central construct of Reiter's logic – are viewed as models for certain formulas encoding the existence of default derivations.

Our approach has certain commonalities with the Nerode, Marek, and Remmel theory, in that we view extensions model-theoretically. However, we use extensions as models for ordinary first-order logic, not the special logic used by Marek, Nerode and Remmel. It will also be clear that first-order logic is not the only possible logic for which default models could serve as a semantic space. But we concentrate on the first-order case, since that involves the use of constraints.

Our treatment also has the advantage that one can analyze default reasoning situations by working directly with models, as one does all the time in ordinary mathematical reasoning. This contrasts with the approaches of MNR and Etherington, where in the first case, the logic describes a proof theory, and in the second, where one works with sets of first order models as models for default logic. We hold the thesis that Reiter's default systems should be regarded, not as proof rules, but as *algorithms for the direct construction of partial models for some appropriate logic*. This is a simple and radical reconstruction of default reasoning. To give it a proper explication, though, we use domain-theoretic tools – information systems and Scott domains, in particular, since in our view default reasoning is about what happens when information necessary to resolve a question is lacking. No one to date (including us) has attempted to use the full power of domain theory to attack such questions.

As we have stated, we want our models to be governed by *constraints*, which in our setting are thought of as laws which govern the behavior of partial models, but which are in the background. This is an idea taken from situation

*Research sponsored by NSF Grant IRI-9120851.

theory; see Barwise's book [2], but it is very much akin to the notion of constraint in CLP. We encode a constraint theory into the monotonic forcing relation $\vdash$ of a Scott information system appropriate for a first-order logic semantics.

How to accomplish this encoding is not absolutely clear. One possibility is to use a generalization of information systems themselves, due to Saraswat *et al.* [11], to the first-order case. We have determined, however, that such a move is unneccessary. We represent constraint theories as a special case of ordinary monotonic information systems.

The next problem is how to add a non-monotonic component to information systems. This we have done by simply adding default forcing rules to Scott's systems. We prove in the full paper that this a natural move, not an *ad hoc* one, by showing that (normal) default information systems characterize an abstract domain-theoretic concept: the idea of a *normal covering relation* on a Scott domain. This result generalizes Scott's own characterization of domains via information systems.

A final problem is how to use the domains we generate as models for first-order logic itself, and specifically, how to interpret negation. We have chosen a restricted, positive version of first-order logic, which only allows negation on atomic formulas. Then we introduce a notion $\mid\!\sim$ of *non-monotonic consequence* between sentences of first-order logic, as in Kraus, Lehmann, and Magidor [6]. We say that in a default model structure, one sentence non-monotonically entails a second if the second holds in all extensions of "small" partial models of the first. Here "smallness" is interpreted with respect to the natural partial order associated with a Scott domain.

We then turn to the construction of probabilities using finite models. Finite default models are of course a special case of our theory. We can generalize the usual finite model theory to partial models, and can use default rules to assign probabilities to statements in FOL, representing an agent's degree of belief in certain situations obtaining. This gives a way of thinking about the usual "birds normally fly" as a probabilistic statement. We illustrate this method in the resolution of an anomaly with standard default reasoning, due to Poole [9].

The paper is organized as follows. In section 2 we cover the basics of domain theory and information systems, introduce our non-monotonic generalization, and state a representation theorem for default domains. In Section 3 we show how to interpret first-order positive logic using default models. This is where constraints play a crucial role. Then in Section 4 we introduce our notion of conditional degree of belief, and treat Poole's anomaly. Finally, in Section 5 we mention other applications of our theory, including some speculations on CLP itself.

2 Default Domain Theory

2.1 Domain Theory

To make the paper self-contained, we recall some basic domain theoretic definitions. A directed subset of a partial order $(D, \sqsubseteq)$ is a non-empty set $X \subseteq D$ such that for every $x, y \in X$, there is a $z \in X$, $x \sqsubseteq z$ & $y \sqsubseteq z$. A complete partial order (cpo) is a partial order which has a bottom element and least upper bounds of directed sets. A subset $X \subseteq D$ is bounded (or compatible, consistent) if it has an upper bound in D. An isolated (or compact, finite) element x of D is one such that whenever $x \sqsubseteq \bigsqcup X$ with X directed, we also have $x \sqsubseteq y$ for some $y \in X$. A cpo is algebraic if each element of which is the least upper bound of a set of isolated elements. A cpo is ω-algebraic if it is algebraic and the set of isolated elements is countable. A Scott domain is an ω-algebraic cpo in which every compatible subset has a least upper bound. By convention, we write $x \uparrow y$ if the set $\{x, y\}$ is bounded.

Now we review Scott's representation of domains using *information systems*, which can be thought of as general concrete monotonic "rule systems" for building Scott domains.

Definition 2.1 *An information system is a structure* $\underline{A} = (A, Con, \vdash)$ *where*

- A *is a countable set of tokens,*
- $Con \subseteq \mathrm{Fin}(A)$, *the consistent sets,*
- $\vdash \subseteq Con \times A$, *the entailment relation,*

which satisfy

1. $X \subseteq Y$ & $Y \in Con \Rightarrow X \in Con$,
2. $a \in A \Rightarrow \{a\} \in Con$,
3. $X \vdash a$ & $X \in Con \Rightarrow X \cup \{a\} \in Con$,
4. $a \in X$ & $X \in Con \Rightarrow X \vdash a$,
5. $(\forall b \in Y.\, X \vdash b$ & $Y \vdash c) \Rightarrow X \vdash c$.

Example: Approximate real numbers. For tokens, take the set A of pairs of rationals $\langle q, r\rangle$, with $q \leq r$.

The idea is that a pair of rationals stands for the "proposition" that a yet to be determined *real number* is in the interval $[q, r]$ whose endpoints are given by the pair.

Define a finite set X of "intervals" to be in in Con if X is empty, or if the intersection of the "intervals" in X is nonempty. Then say that a set $X \vdash \langle q, r\rangle$ iff the intersection of all "intervals" in X is contained in the interval $[q, r]$. Note that there is only atomic structure to these propositions. We cannot negate them or disjoin them.

The representation of Scott domains uses the auxiliary construct of ideal elements.

Definition 2.2 *An (ideal) element of an information system $\underline{A}$ is a set x of tokens which is*

1. *consistent:* $X \subseteq x \Rightarrow X \in Con$,
2. *closed under entailment:* $X \subseteq x \;\&\; X \vdash a \Rightarrow a \in x$.

The collection of ideal elements of $\underline{A}$ is written $|\underline{A}|$.

Example. The ideal elements in our approximate real system are in 1-1 correspondence with the collection of closed real intervals $[x, y]$ with $x \leq y$. Although the collection of ideal elements is partially ordered by inclusion, the domain being described – intervals of reals – is partially ordered by reverse interval inclusion. The total or maximal elements in the domain correspond to "perfect" reals $[x, x]$. The bottom element is a special interval $(-\infty, \infty)$.

It can be easily checked that for any information system, the collection of ideal elements ordered by set inclusion forms a Scott domain. Conversely, every Scott domain is isomorphic to a domain of such ideal elements. These results are basic in domain theory, and have been generalized to other classes of complete partial orders by Zhang [12] and others.

2.2 Default Information Systems

We generalize the theory of information systems by simply adding a default component. We should mention at this point that we limit ourselves to the so-called *normal* default structures. The reason for this is not that we cannot define general default rules, but rather that there are problems with the existence of extensions in the full case that we want to avoid.

Definition 2.3 *A normal default information structure is a tuple*

$$\underline{A} = (A, Con, \Delta, \vdash)$$

where $(A, Con, \vdash)$ is an information system, Δ is a set of pairs (X, Y) of consistent finite subsets of A, each element of which is written as $\dfrac{X : Y}{Y}$.

In our application, tokens will be "tuples" or *infons* of the form

$$\langle\!\langle \sigma, m_1, \ldots, m_n; i \rangle\!\rangle,$$

where σ is a relation name, the m_j are elements of a structure, and i is a "polarity" – either 0 or 1. The rules in Δ should therefore be read as follows. If the set of tuples X is in our current database, and if adding Y would not violate any database constraints, then add Y.

In default logic, the main concept is the idea of an extension. We define extensions in default model theory using Reiter's conditions, but extensions are now (partial) models. The following definition is just a reformulation, in information-theoretic terms, of Reiter's own notion of extension in default logic.

Definition 2.4 *Let $\underline{A} = (A, \Delta, \vdash)$ be a default information structure, and x a member of $|\underline{A}|$. For any subset S, define $\Phi(x, S)$ to be the union $\bigcup_{i \in \omega} \phi(x, S, i)$, where*

$$\phi(x, S, 0) = x,$$
$$\phi(x, S, i+1) = \overline{\phi(x, S, i)} \cup \bigcup \{Y \mid \frac{X : Y}{Y} \in \Delta \;\&\; X \subseteq \phi(x, S, i) \;\&\; Y \cup S \in Con\}.$$

y is an extension of x if $\Phi(x, y) = y$. In this case we also write $x \delta_{\underline{A}} y$, with the subscript omitted from time to time.

Example (1): The eight queens problem. We have in mind in an 8×8 chessboard, so let $8 = \{0, 1, \ldots, 7\}$. Our token set A will be 8×8. A subset X of A will be in Con if it corresponds to an admissible placement of up to 8 queens on the board. For defaults Δ we take the singleton sets

$$\{\frac{:\{(i,j)\}}{\{(i,j)\}} \mid (i,j) \in 8 \times 8\}.$$

We may take $\vdash$ to be trivial: $X \vdash (i,j)$ iff $(i,j) \in X$. Now, if x is an admissible placement, then the extensions y of x are those admissible placements containing x and so that no more queens may be placed without violating the constraints of the problem.

Example (2): Default approximate reals. Use the information system described above. We might like to say that "by default, a real number is either between 0 and 1, or is the number π". We could express this by letting Δ consist of the rules $\frac{:Y}{Y}$, where Y ranges over singleton sets of rational pairs $\{\langle p, q\rangle\}$ such that $p \leq 0$ and $q \geq 1$, together with those pairs $\{\langle r, s\rangle\}$ such that $r < \pi$ and $s > \pi$. Then, in the ideal domain, the only extension of $[-1, 2]$ would be $[0, 1]$; the interval $[-2, 0.5]$ would have $[0, 0.5]$ as an extension, and there would be 2 extensions of $[-2, 4]$, namely $[0, 1]$ and $[\pi, \pi]$.

These examples are intended to wean the reader away from the view of defaults as default logic. In the eight queens problem, it seems desirable to have a language for reasoning about differing placements. For example, given a placement x, is there an extension y which uses all eight queens? This corresponds exactly to our philosophy: default systems are used model-theoretically, and logic is used to describe default models. The example also foreshadows how constraints are to be modeled using the $\vdash$ relation. For first-order systems the constraints can be explicitly described as first-order axioms.

2.3 Representation theory

We now restate the definition of extension in domain-theoretic terms. In the following, D^0 is the set of finite elements of the Scott domain D.

Definition 2.5 *Let $(D, \sqsubseteq)$ be a Scott domain, and Λ a subset of $D^0 \times D^0$. The normal preferential cover δ_Λ determined by Λ is the binary relation on D given by the condition*

$$x \delta_\Lambda y \iff y = x \sqcup \bigsqcup \{\mu \mid (\exists \lambda \sqsubseteq y) : ((\lambda, \mu) \in \Lambda \ \& \ \mu \uparrow y\}.$$

Here $\mu \uparrow y$ means that the set $\{\mu, y\}$ is bounded or consistent.

The idea of this definition is that $\langle \lambda, \mu \rangle \in \Lambda$ denotes a default of the form $\frac{X : Y}{Y}$. One can prove all sorts of facts about extensions purely order-theoretically, assuming that D is a Scott domain.

Theorem 2.1 *Let D be a Scott domain and Λ be as above.. We have*

1. $\forall x \in D \exists y \in D \ x \delta_\Lambda y$.
2. *If $x \delta_\Lambda y$ then $y \sqsupseteq x$.*
3. *$x \delta_\Lambda y$ and $y \delta_\Lambda z$ implies $y = z$.*
4. *If $x \delta_\Lambda y$ and $x \sqsubseteq z \sqsubseteq y$ then $x \delta_\Lambda y$.*
5. *If $x \delta_\Lambda y$ and $x \delta_\Lambda y'$ then either $y = y'$ or $y \not\uparrow y'$.*

For the special case of *precondition-free* normal defaults, where $\langle \lambda, \mu \rangle \in \Lambda \rightarrow \lambda = \bot$, we have

Theorem 2.2 *$x \delta_\Lambda y$ if and only if there is a B, a maximal subset of $\{\mu \mid \langle \bot, \mu \rangle \in \Lambda\}$ such that $B \cup \{x\}$ is bounded, and such that*

$$y = x \sqcup \bigsqcup \{\mu \mid \mu \in B\}$$

These results all suggest a general representation theorem.

Theorem 2.3 *Every normal preferential covering relation on a Scott domain is isomorphic to the extension relation generated by a normal default system, and conversely.*

The real point of the theorem is that, since Scott domains are quintessentially model-theoretic, we should regard default systems this way too.

3 Constraint default structures for first-order logic

Assume, for purposes of this paper, that we are given a signature for first-order logic with equality, and with no function symbols other than constants. (This is essentially Datalog.) We will interpret first order logic using a nonstandard class of models. Our structures will be default information systems based on a particular set of individuals M. We first have to assume some *constraints* on any relations which are going to be holding in such sets M. These constraints will be used to generate the monotonic forcing relation $\vdash$ in the default structure. (The defaults themselves can be arbitrary, as long as they are normal.) We can use sets C of arbitrary closed formulas of first-order logic to state background constraints; in fact, we can use any language for which first-order structures are appropriate models.

To interpret formulas, we first of all choose some set M of individuals. We do *not* fix relations on M as in the standard first-order case, but we do choose particular individuals to interpret the constants[1]. Now, tokens will be infons of the form

$$\sigma = \langle\langle R, m_1, \ldots, m_n; i \rangle\rangle$$

where R is a relation name, $m_j \in M$, and $i \in \{0, 1\}$. (This last item is called the *polarity* of the token.) We say that a set s of these tokens is *admissible* if (i) it does not contain any tokens conflicting in polarity, and (ii) it matches a model of C in the usual first-order sense. That is, there is a structure

$$\mathcal{M} = (M, (R_1, \ldots R_k))$$

where the R_j are relations on M of the appropriate arities, such that $\mathcal{M}$ is a model of C, and such that

$$\langle\langle R_j, m_1, \ldots, m_n; 1 \rangle\rangle \in s \Rightarrow R_j(m_1, \ldots, m_n) \text{ is true in } \mathcal{M}.$$

Similarly,

$$\langle\langle R_j, m_1, \ldots, m_n; 0 \rangle\rangle \in s \Rightarrow R_j(m_1, \ldots, m_n) \text{ is false.}$$

An admissible set of infons is *total* if it is maximal in the subset ordering on sets of infons. A total set is isomorphic to an ordinary first-order structure $\mathcal{M}$.

Now we can specify a default information structure relative to M and C. Actually, the work is in specifying the strict (monotonic) part of the system. The defaults can be arbitrary normal ones.

Definition 3.1 *Let M be a set, and C a constraint set. A first-order default information structure relative to M and C is a structure of the form*

$$\underline{A}(M, C) = (A, \mathit{Con}, \Delta, \vdash)$$

where A is the token set described above. A finite set X of tokens will be in Con if it is admissible, and $X \vdash \sigma$ iff for any total admissible set t, if $X \subseteq t$ then $\sigma \in t$.

Examples. The above definition encodes the constraints C into the $\vdash$ relation of the information system. For example, consider the constraint obtained by taking C to be the true formula t. Intuitively, this should be no constraint at all, so our entailment relation should be the minimal one in which $X \vdash \sigma$ if and only if $\sigma \in X$. This is in fact the case. First notice that because $C = \mathrm{t}$, that a total admissible set t is one which (i) contains no infon $\sigma = \langle\langle \sigma, m; i \rangle\rangle$ and the *dual* infon $\overline{\sigma}$ of opposite polarity; and (ii) for any infon σ, contains either σ or $\overline{\sigma}$. Now let X be a finite set of infons. If $X \vdash \sigma$ then by properties of information systems, the dual infon $\overline{\sigma} \notin X$. By definition of $\vdash$, for any total admissible set t of infons, if $X \subseteq t$ then $\sigma \in t$. If σ is not in X, let t be a total admissible set containing X and the infon $\overline{\sigma}$ of opposite polarity. Then both σ and $\overline{\sigma}$ would be in t, which is not possible for an admissible set.

Here is a more interesting constraint system:

[1] In terms of philosophy of language, we are taking constants to be rigid designators.

Exercise. Consider the eight queens problem again. Write first-order constraints C expressing the constraints of the eight queens problem.

Notice that our general definition is easily modified to particular classes of interpretations. For example, our constraints may be stated for just one intended model, say the real numbers with addition and multiplication. In that case, we choose our sets M to be allowable by the particular interpretation class, and we change the definition of admissibility so that first-order structures are chosen from our particular class as well. Technically, we should restrict M to be countable, so that our Scott domain is in fact ω-algebraic. In fact, though, we will mostly be interested in *finite* default models for first-order logic.

3.1 Syntax and Semantics

In this workshop paper, we give just one application of constraints in first-order default model theory. This consists of a probabilistic resolution of an anomaly in default logic, due to Poole [9]. We begin, though, with the syntax and semantics of first-order logic itself.

Here is the official syntax:

$$\varphi ::= \mathrm{t} \mid \mathrm{f} \mid R(v_1, \ldots, v_n) \mid \neg R(v_1, \ldots, v_n) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \exists x \varphi \mid \forall x \varphi$$

where $R(v_1, \ldots, v_n)$ is an atomic formula and the v_i are either variables in some set X, or constants.

We remark that one can treat the case of arbitrary negations by interpreting negation using the strong Kleene truth tables. This interpretation gives us a persistence property for partial models that is useful in establishing properties of nonmonotonic entailment and conditional degree of belief. For simplicity we omit this general definition.

Next we turn to the semantics. Fix constraints C and a set M. Then select a default information structure $\underline{A}(M, C)$. Let s be an ideal element (also called a situation) of $\underline{A}(M, C)$, and let $\alpha : X \to M$.

Then the clauses for $(s, \alpha) \models \varphi$ are as follows.

- $(s, \alpha) \models \mathrm{t}$ always and f never;
- $(s, \alpha) \models R(v_1, \ldots, v_n)$ iff $\langle\!\langle R, \alpha(v_1), \ldots, \alpha(v_n); 1 \rangle\!\rangle \in s$. (If some of the v_i are constants, we use instead of $\alpha(v_i)$ the fixed interpretation of that constant.)
- $(s, \alpha) \models \neg R(v_1, \ldots, v_n)$ iff $\langle\!\langle R, \alpha(v_1), \ldots, \alpha(v_n); 0 \rangle\!\rangle \in s$;
- The usual clauses for $\wedge$ and $\vee$;
- $(s, \alpha) \models \exists v \varphi$ iff for some $m \in M$, $(s, \alpha[v \leftarrow m]) \models \varphi$;
- $(s, \alpha) \models \forall v \varphi$ iff for all $m \in M$, $(s, \alpha[v \leftarrow m]) \models \varphi$.

3.2 Nonmonotonic consequence

Our semantics can now be used to define a relation of nonmonotonic entailment, written $\mid\!\sim$, between sentences of our (positive) first-order logic. Understanding this notion is a step towards understanding the probabilistic measure introduced in the next subsection.

Intuitively, when we say that φ nonmonotonically entails ψ, we mean that having only the information φ, we can "leap to the conclusion" ψ. The usual example is that, knowing only that Tweety is a bird, we can leap to the conclusion that Tweety flies, even though penguins do not fly. A great deal of effort in the AI community has gone into giving a proper interpretation to the assertion $\varphi \mid\!\sim \psi$. We use (finite) default models and extensions to interpret it.

The notion of "only knowing" φ in $\varphi \mid\!\sim \psi$ [7], given a default information structure, is captured by interpreting the antecedent φ in a certain small class of situations for the structure. There are at least two possibilities for this class. One natural one is to use all set-theoretically minimal situations supporting φ. The second is to interpret φ in the *supremum closure* of these minimal models. We choose the second in this paper, because it seems better motivated from the probabilistic standpoint to be given in the next subsection.

We therefore make the following definitions.

Definition 3.2 *Let $\underline{A}(M, C)$ be a default information structure, and φ a sentence of our logic. Let s, t range over situations.*

- *$MM(\varphi)$ is the set $\{s \mid s$ is minimal such that $s \models \varphi\}$;*
- *$U(\varphi)$ is the supremum closure of $MM(\varphi)$: the collection of situations obtained by taking consistent least upper bounds of arbitrary subcollections of $MM(\varphi)$. If $s \in U(\varphi)$ we will say that s is a minimal-closure model of φ.*

Notice that since our logic is positive, every situation in $U(\varphi)$ will support φ.

Given these concepts, we can define nonmonotonic consequence as follows.

Definition 3.3 *Let φ and β be sentences in first-order logic. Let $\underline{A} = A(M, C)$ be a finite normal default information system as above. We say that $\varphi \mid\sim_{\underline{A}} \beta$ if for all minimal-closure models $s \in U(\varphi)$,*

$$\forall t : t \text{ is an } \underline{A}\text{-extension of } s \Rightarrow t \models \beta.$$

Example. We give the standard bird-penguin example. Assume that our language contains two predicates $Bird$ and $Penguin$, and that $Tweety$ is a constant. Let C be the constraint

$$(\forall x)(Penguin(x) \rightarrow Bird(x) \wedge \neg Fly(x)).$$

Consider a structure $A(M, C)$. Form the defaults

$$\frac{\langle\langle bird, m; 1\rangle\rangle : \langle\langle fly, m; 1\rangle\rangle}{\langle\langle fly, m; 1\rangle\rangle}$$

for each m in M. These defaults express the rule that birds normally fly. We then have

$$MM(Bird(Tweety)) = U(Bird(Tweety)) = \{\{\langle\langle Bird, tw; 1\rangle\rangle\}\}$$

where tw is the element of M interpreting $Tweety$.

The only extension of $\{\langle\langle Bird, tw, ; 1\rangle\rangle\}$ is $\{\langle\langle bird, tw; 1\rangle\rangle, \langle\langle fly, tw; 1\rangle\rangle\}$. Therefore

$$Bird(Tweety) \mid\sim Fly(Tweety).$$

We do not have $Penguin(Tweety) \mid\sim Fly(Tweety)$, because of the constraint C.

4 Probabilities, Constraints, and Default Models

One of the most interesting ramifications of our approach is that we can use defaults to generate *degrees of belief* or *subjective probabilities* of various logical statements. By "subjective probability" we mean an analogue of the usual probability, a number that would be assigned to a statement by a particular agent or subject, given a default system and some basic constraints on the world. Let us illustrate with an example of Poole [9].

4.1 Poole's anomaly

Assume that there are exactly three mutually exclusive types of birds: penguins, hummingbirds, and sandpipers. It is known that penguins don't fly, that hummingbirds are not big, and that sandpipers don't nest in trees. Now suppose we want to assert that the typical bird flies. Since we only speak of birds, we can do this with the precondition-free "open default"

$$\frac{: fly(x)}{fly(x)}.$$

(This notation is of course a schema for the model-theoretic defaults in the default information structure to be given below.) We would also like to say that the typical bird is (fairly) big, and that it nests in a tree. Similar defaults are constructed to express these beliefs.

The " paradox" is that it is impossible now to believe that Tweety, the typical bird, flies. To see why, let us formalize the problem more fully in our first-order language. Let C_1 be the obvious first-order sentence asserting that every individual is one of the three types of birds, and that no individual is of more than one type. Let C_2 be the conjunction of three sentences expressing abnormalities. One of these is, for example,

$$\forall x(Penguin(x) \rightarrow \neg fly(x)).$$

Let the background constraint C be $C_1 \wedge C_2$.

The defaults must be given in the semantics. Let M be a finite set, and consider the first-order default structure A with precondition-free defaults

$$\frac{: \langle\!\langle fly, m; 1 \rangle\!\rangle}{\langle\!\langle fly, m; 1 \rangle\!\rangle}; \frac{: \langle\!\langle big, m; 1 \rangle\!\rangle}{\langle\!\langle big, m; 1 \rangle\!\rangle}; \frac{: \langle\!\langle tree, m; 1 \rangle\!\rangle}{\langle\!\langle tree, m; 1 \rangle\!\rangle}.$$

We need only precondition-free defaults, because we only speak about birds. Further, we need no infons mentioning penguins, or any of the other species. The constraints can still operate.

We assert that if M has n elements, then there are 3^n extensions of the empty set (which is in fact the least model of t). This is because any extension will include, for each bird $m \in M$, exactly two out of the three infons $\langle\!\langle fly, m; 1 \rangle\!\rangle$, $\langle\!\langle big, m; 1 \rangle\!\rangle$, $\langle\!\langle tree, m; 1 \rangle\!\rangle$. The extension cannot contain all three infons, because the constraints rule that out. So each of n birds has three choices, leading to 3^n extensions.

One such extension is

$$\{\langle\!\langle big, m; 1 \rangle\!\rangle : m \in M\} \cup \{\langle\!\langle tree, m; 1 \rangle\!\rangle : m \in M\}$$

which omits any infons of the form $\langle\!\langle fly, m; 1 \rangle\!\rangle$. This extension is one where *no* birds fly, where all birds are penguin-like. So now if *Tweety* is a constant of our language, then the formula *Fly(Tweety)* is not a nonmonotonic consequence of the "true" formula true, whose minimal model is the empty set. Further, if we move to the situation of seventeen bird types, each with its own distinguishing feature, we still have the case that Tweety cannot be believed to be flying. Poole suggests that this raises a problem for most default reasoning systems.

4.2 A probabilistic solution

We now contend that the problem is not so severe. Notice that it is only in 3^{n-1} extensions that Tweety does not fly. This is because in an extension where Tweety does not fly, the infons involving Tweety must assert that he is big and lives in a tree. Tweety thus only has one non-flying choice. The other $n - 1$ birds have the same three choices as before. It seems therefore truthful to say that with probability $(3^n - 3^{n-1})/3^n = 2/3$, Tweety believably does fly. Moreover, imagine a scenario with seventeen mutually exclusive bird types, the same kinds of exceptions for each type, and defaults for all of the types. Then we would get that Tweety flies with probability 16/17.

We use this example to define our notion of subjective probability:

Definition 4.1 *Let φ be a sentence of positive first-order logic. Assume that relative to a constraint C, φ has minimal models in a structure M with n elements. Then the* **conditional subjective probability** $Pr([\psi \mid \varphi]; n, C)$ *is defined to be the quantity*

$$\frac{1}{N} \sum_{s \in U(\varphi)} \frac{card\{e : s\ \delta\ e\ \&\ e \models \psi\}}{card\{e : s\ \delta\ e\}}$$

where N is the cardinality of $U(\varphi)$.

Example. Referring to the case of Tweety above, we have

$$Pr([Fly(Tweety) \mid t]; n, C) = 2/3.$$

Note the non-dependence on n. We conjecture that there is in fact a limit law, as in the case of 0-1 laws for first-order systems, in operation here. (See the survey by Compton [3] for a general introduction to 0-1 laws.)

Our definition bears a strong resemblance to the notion defined by Bacchus, Grove, Halpern, and Koller [1]. Their definition of the conditional probability of a statement ψ given another statement φ, though, is not made with reference to a given default information system. Instead, defaults are "translated" into a special logic for reasoning about statistical information. (For example, one can say that the proportion of flying birds out of all birds is approximately .9). Then, the translated default statements, and the given formulas φ and ψ are given a conditional probability in a standard first-order structure, much as in the case of 0-1 laws[2]. Our corresponding "translation" of default statements is into a system of default rules, just as in Reiter's formulation.

Our semantics also contrasts with that of BGHK in that it looks at partial worlds as well as total ones, and can assign probabilities to a statement's not being resolved one way or another.

[2]Grove, Halpern, and Koller prove limit laws for their notion of conditional probability in [5].

We illustrate the role of partial models, and the role played by the class $U(\varphi)$, with an example not involving constraints or default rules at all.

Consider the "evidence" $\varphi = (\exists x)P(x)$, where P is some unary predicate. What is the degree of belief in $P(tweety)$, given φ? Assume a set M with n individuals. From the definitions, we see that $U(\varphi)$ can consist of any nonempty subset of the set

$$\{\langle\langle p, m; 1\rangle\rangle \mid m \in M\}.$$

Thus $N = 2^n - 1$, and

$$Pr([P(Tweety) \mid \varphi]; n, t) = \frac{1}{2^n - 1} \cdot card\{s \in U(\phi) \mid \langle\langle P, tweety; 1\rangle\rangle \in s\}.$$

There are 2^{n-1} sets of infons containing the desired one, so that our answer is

$$\frac{1}{2 - 1/2^{n-1}}$$

which is close to $1/2$ for large n.

This result agrees with the BGHK definition of conditional probability, but the analogous result for the negative case does not. In the standard case, every question is settled in a structure. But in our case, we do not have this. For example $\neg P(Tweety)$ is not settled in any situation in $U(\varphi)$ above, because $\langle\langle P, m; 0\rangle\rangle \notin U(\varphi)$ for any m. Thus we get

$$Pr([\neg P(Tweety) \mid \varphi]; n, t) = 0$$

whereas BH would get a number slightly less than $1/2$.

Finally, our semantics avoids certain "dependencies on language." Referring to Poole's example, notice that if we change our set of predicates and constraints to the case of seventeen bird types, but retain the rule system for three types, then we still get the same degree of belief (2/3) for Tweety's flying. The BGHK semantics does not have this property. In our case, it is as if, when we expand our universe to the case of seventeen types, we have not yet learned the new rules for building models of those types of birds.

As a general comparison of our method with that of BGHK, it seems fair to say that we are approaching the problem of re-creating direct inference (reasoning logically from statistical information) by using synthetic rule-based systems with constraints, whereas BGHK are concerned with the analytic reasons for the rules in the first place. In any case, the connections deserve futher exploration.

5 Conclusion

We hope to have shown in this paper some of the uses of constraints in non-monotonic rule-based reasoning, embodied in default model theory. There should be many more interesting applications of our technique. One such may be to logic programming and CLP itself. The connections between default logic and logic programs with negation as failure are well known. Our research indicates that default rules should be regarded as algorithms to construct models. If this is so, then by analogy, logic programs should be constructing models too. (We think that this is exactly the new view of constraint programming.) Then, we would need a new logic to reason about the worlds (in fact partial worlds) built by a logic program, which might be rechristened a "model generator." But this is exactly the view of denotational semantics and standard logics of programs: programs are interpreted model-theoretically, and a specification logic is used to reason about the models that have been "built" by the program. These analogies and questions are what we want to work on in the near future. We hope to have awakened interest in the topic for the practitioners of constraint programming.

References

[1] F. Bacchus, A. Grove, J. Halpern, and D. Koller. Statistical foundations for default reasoning. IJCAI 1993, to appear.

[2] Jon Barwise. *The Situation in Logic*. 17. Center for Study of Language and Information, Stanford, California, 1989.

[3] K. J. Compton. 0-1 laws in logic and combinatorics. In I. Rival, editor, *NATO Adv. Study Inst. on Algorithms and Order*, pages 353–383, 1988.

[4] D. W. Etherington. *Reasoning with incomplete information*. Research Notes in Artificial Intelligence. Morgan Kaufman, 1988.

[5] A. Grove, J. Halpern, and D. Koller. Asymptotic conditional probabilities for first-order logic. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 294–305, 1992.

[6] S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models, and cumulative logics. *Artificial Intelligence*, 44:167–207, 1990.

[7] H. J. Levesque. All i know: A study in autoepistemic logic. *Artificial Intelligence*, 42:263–309, 1990.

[8] W. Marek, A. Nerode, and J. Remmel. A theory of nonmonotonic rule systems. In *Proceedings of 5th IEEE Symposium on Logic in Computer Science*, pages 79–94, 1990.

[9] D. L. Poole. What the lottery paradox tells us about default reasoning. In *Proceedings of First Annual Conference on Knowledge Representation*. Morgan Kaufmann, 1989.

[10] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[11] V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of POPL 1991*, 1991.

[12] Guo-Qiang Zhang. *Logic of Domains*. Birkhauser, Boston, 1991.

The SkyBlue Constraint Solver and Its Applications

Michael Sannella
Department of Computer Science
and Engineering, FR-35
University of Washington
Seattle, WA 98195
sannella@cs.washington.edu

Abstract

The SkyBlue constraint solver is an efficient incremental algorithm that uses local propagation to maintain sets of required and preferential constraints. SkyBlue is a successor to the DeltaBlue algorithm, which was used as the constraint solver in the ThingLab II user interface development environment. Like DeltaBlue, SkyBlue represents constraints between variables by sets of short procedures (*methods*) and incrementally resatisfies the set of constraints as individual constraints are added and removed. DeltaBlue has two significant limitations: cycles of constraints are prohibited, and constraint methods can only have a single output variable. SkyBlue relaxes these restrictions, allowing cycles of constraints to be constructed (although SkyBlue may not be able to satisfy all of the constraints in a cycle) and supporting multi-output methods. This paper presents the SkyBlue algorithm and discusses several applications that have been built using SkyBlue.

1 Introduction

The DeltaBlue algorithm is an incremental algorithm for maintaining sets of required and preferential constraints (constraint hierarchies) using local propagation [4, 5, 9]. The ThingLab II user interface development environment was based on DeltaBlue, demonstrating its feasibility for constructing user interfaces [5]. However, DeltaBlue has two significant limitations: cycles in the graph of constraints and variables are prohibited (if a cycle is found, an error is signaled and the cycle is broken by removing a constraint) and constraint methods can only have one output variable.

The SkyBlue algorithm was developed to remove these limitations. Even though it is not always possible to solve cycles of constraints using local propagation, SkyBlue allows constructing such cycles. SkyBlue cannot satisfy the constraints around a cycle, but it correctly maintains the non-cyclic constraints elsewhere in the graph. In some situations it may be possible to solve the subgraph containing the cycle by calling a more powerful solver. Future work will extend SkyBlue to call specialized constraint solvers to solve the constraints around a cycle, and continue using local propagation to satisfy the rest of the constraints.

SkyBlue also supports constraints with multi-output methods, which are useful in many situations. For example, suppose the variables X and Y represent the Cartesian coordinates of a point, and the variables ρ and θ represent the polar coordinates of this same point. To keep these two representations consistent, one would like to define a constraint with a two-output method $(X, Y) \leftarrow (\rho \cos\theta, \rho \sin\theta)$ and another two-output method in the other direction $(\rho, \theta) \leftarrow (\sqrt{X^2 + Y^2}, \arctan(Y, X))$. Multi-output methods are also useful for accessing the elements of compound data structures. For example, one could unpack a compound *CartesianPoint* object into two variables using a constraint with methods $(X, Y) \leftarrow (Point.X, Point.Y)$ and $Point \leftarrow CreatePoint(X, Y)$.

Support for multi-output methods introduces a performance issue. It has been proved that supporting multi-output methods is NP-complete [5]. In actual use, the worst-case time complexity has not been a problem. Both DeltaBlue and SkyBlue typically change only a small subgraph of the constraint graph when a constraint is added or removed so the actual performance is usually sub-linear in the number of constraints. Over the same types of constraint graphs that DeltaBlue can handle (no cycles, single-output methods), SkyBlue has been measured at about half the speed of DeltaBlue. In the future SkyBlue may be

extended to detect when the constraint graph contains no cycles or multi-output methods and achieve the speed of DeltaBlue in this case.

SkyBlue is currently being used as the constraint solver in several applications (see Section 6). SkyBlue implementations are available from the author. Detailed information on SkyBlue, including complete pseudocode, is available in a technical report [7].

2 Method Graphs

A SkyBlue constraint is represented by one or more *methods*. Each method is a procedure that reads the values of a subset of the constraint's variables (the method's *input variables*) and calculates values for the remaining variables (the method's *output variables*) that satisfy the constraint. For example, the constraint $A + B = C$ could be represented by three methods: $C \leftarrow A + B$, $A \leftarrow C - B$, and $B \leftarrow C - A$. If the value of A or B were changed, SkyBlue could maintain the constraint by executing $C \leftarrow A + B$ to calculate a new value for C.

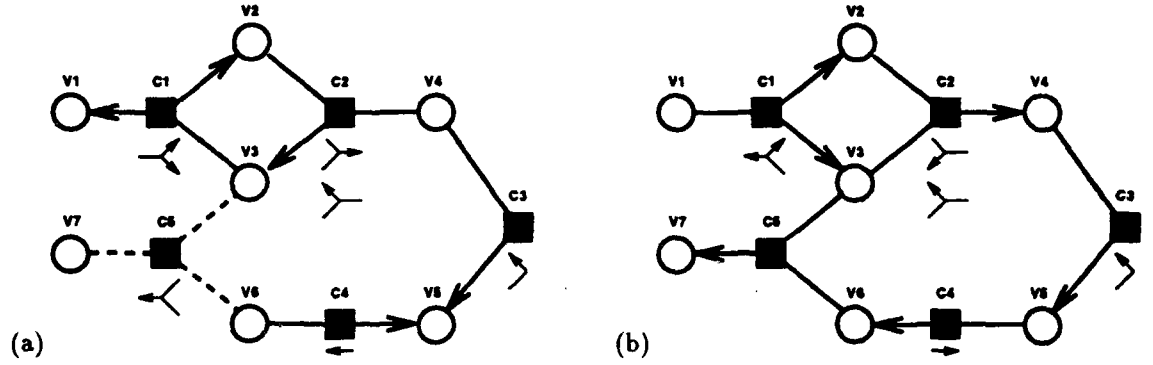


Figure 1: (a) A method graph with an unenforced constraint ($C5$), a method conflict (at $V5$), and a directed cycle (between $C1$ and $C2$). (b) Another method graph for the same constraints where all of the constraints can be satisfied.

To satisfy a set of constraints, SkyBlue chooses one method to execute from each constraint, known as the *selected method* of the constraint. The set of constraints and variables form an undirected *constraint graph* with edges between each constraint and its variables. The constraint graph, together with the selected methods, form a directed *method graph*. In this paper, method graphs are drawn with circles representing variables and squares representing constraints (Figure 1). Lines are drawn between each constraint and its variables. If a constraint has a selected method, arrows indicate the outputs of the selected method. If a constraint has no selected method, it is linked to its variables with dashed lines. Small diagrams beneath each constraint square indicate the unselected methods for the constraint (if any). These diagrams are particularly useful when a constraint doesn't have methods in all possible directions or has multi-output methods (such as $C1$).

The following terminology will be used in this paper. If a constraint has a selected method in a method graph the constraint is *enforced* in that method graph, otherwise it is *unenforced*. Assigning a method as the selected method of a constraint is known as *enforcing* the constraint. Assigning no method as the selected method of a constraint is known as *revoking* the constraint. A variable that is an output of a constraint's selected method is *determined* by that constraint. A variable that is not an output of any selected method is *undetermined*. Following the selected method's output arrows leads to *downstream* variables and constraints. Following the arrows in the reverse direction leads to *upstream* variables and constraints.

If a method graph contains two or more selected methods that output to the same variable, this is a *method conflict*. In Figure 1a, there is a method conflict between the selected methods of $C3$ and $C4$. SkyBlue prohibits method conflicts because they prevent satisfying both constraints simultaneously. If we satisfy $C3$ by executing its selected method (setting $V5$), and then satisfy $C4$ by executing its selected method (again setting $V5$), then $C3$ might no longer be satisfied. If a method graph has no method conflicts and no directed cycles, then it can be used to satisfy the enforced constraints by executing the selected

methods so any determined variable is set before it is read (i.e., executing the methods in topological order). For example, Figure 1b shows a method graph for the same constraints where all of the constraints can be satisfied by executing the selected methods for $C1$, $C2$, $C3$, $C4$, and $C5$, in this order. The method graph specifies how to satisfy the enforced constraints, regardless of the particular values of the variables.

If a method graph contains directed cycles, such as the one between $C1$ and $C2$ in Figure 1a, it is not possible to find a topological sort of the selected methods. In this case, SkyBlue sorts and executes only the selected methods upstream of cycles. Any methods in a cycle or downstream of a cycle are not executed and their output variables are marked to specify that their values do not necessarily satisfy the enforced constraints. If a cycle is later broken, the methods in the cycle and downstream are executed correctly. SkyBlue will be extended in the future to call more powerful solvers to find values satisfying a cycle of constraints and then propagate these values downstream.

3 Constraint Hierarchies

An important property of any constraint solver is how it behaves when the set of constraints is overconstrained (i.e., there is no solution that satisfies all of the constraints) or underconstrained (i.e., there are multiple solutions). If the solver is maintaining constraints within a user interface application, it is not acceptable to handle these situations by signaling an error or halting. The *constraint hierarchy* theory presented in [2] provides a way to specify declaratively how a solver should behave in these situations. A constraint hierarchy is a set of constraints, each labeled with a *strength*, indicating how important it is to satisfy each constraint.[1] Given an overconstrained constraint hierarchy, a constraint solver may leave weaker constraints unsatisfied in order to satisfy stronger constraints. If a hierarchy is underconstrained, the solver can choose any solution. The user can control which solution is chosen by adding weak *stay* constraints to specify variables whose value should not be changed.

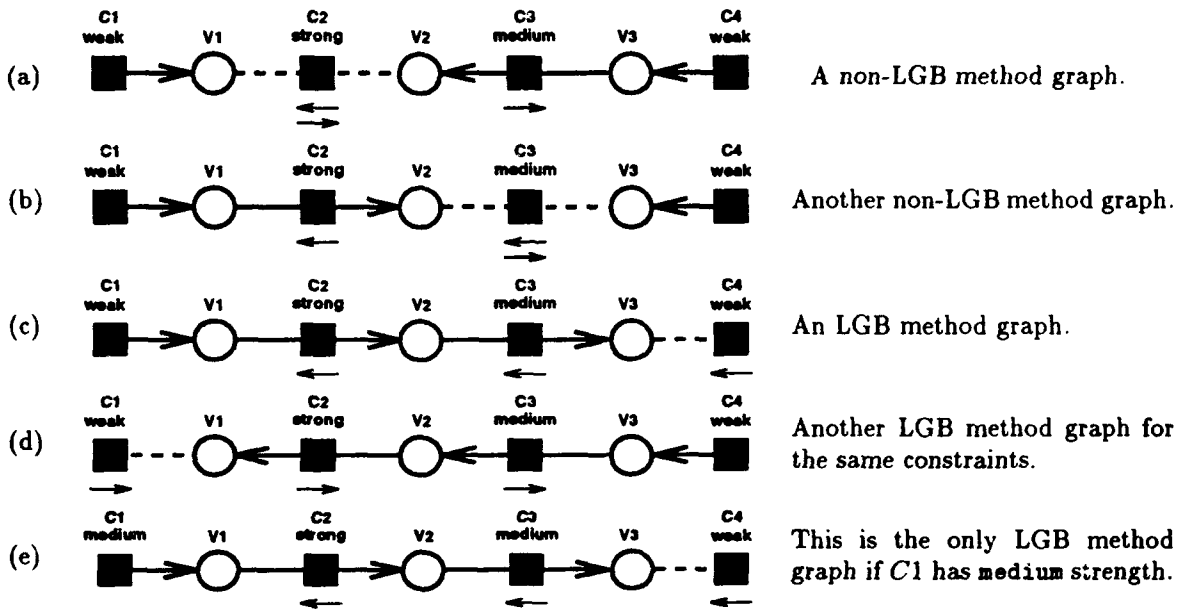


Figure 2: LGB and non-LGB Method Graphs.

The SkyBlue solver uses the constraint strengths to construct *locally-graph-better* (or *LGB*) method graphs [5]. A method graph is LGB if there are no method conflicts and there are no unenforced constraints that could be enforced by revoking one or more weaker constraints (and possibly changing the selected

[1] In this paper, strengths will be written using the symbolic names `required`, `strong`, `medium`, and `weak`, in order from strongest to weakest.

methods for other enforced constraints with the same or stronger strength).[2] For example, consider the method graph in Figure 2a. This graph is not LGB because the **strong** constraint $C2$ could be enforced by choosing the method that outputs to $V2$ and revoking the **medium** constraint $C3$, producing Figure 2b. Actually, this method graph is not LGB either since $C3$ could be enforced by revoking $C4$, producing Figure 2c. This method graph is LGB since the only unenforced constraint ($C4$) cannot be enforced by revoking a weaker constraint.

There may be multiple LGB method graphs for a given constraint graph. Figure 2d shows another LGB method graph which is neither better nor worse than Figure 2c. Given these constraints, SkyBlue would construct one of these two method graphs arbitrarily. The constraint strengths could be modified to favor one alternative over the other. For example, if the strength of $C1$ was changed to **medium**, the only LGB method graph would be the one in Figure 2e. One way for the programmer to control the method graphs constructed is to add *stay* constraints that have a single null method with no inputs and a single output. A stay constraint specifies that its output variable should not be changed. A similar type of constraint is a *set* constraint, which sets its output to a constant value. Set constraints can be used to inject new variable values into a constraint graph. In Figure 2, $C1$ and $C4$ are stay or set constraints.

Reference [2] presents several different ways to define which variable values "best" satisfy a constraint hierarchy. The concept of read-only variables extends this theory to constraints that may not be able to set some of their variables, such as SkyBlue constraints without methods in all possible directions. For many constraint graphs, LGB method graphs compute "locally-predicate-better" solutions to the constraint hierarchy (defined in Reference [2]). Reference [5] examines the relation between LGB method graphs and locally-predicate-better solutions.

4 The SkyBlue Algorithm

The SkyBlue constraint solver maintains the constraints in a constraint graph by constructing an LGB method graph and executing the selected methods in the method graph to satisfy the enforced constraints. Initially, the constraint graph and the corresponding LGB method graph are both empty. SkyBlue is invoked by calling two procedures, **add-constraint** to add a constraint to the constraint graph, and **remove-constraint** to remove a constraint. As constraints are added and removed, SkyBlue incrementally updates the LGB method graph and executes methods to resatisfy the enforced constraints.

The presentation of SkyBlue is divided into several sections. Sections 4.1 and 4.2 present an overview of **add-constraint** and **remove-constraint**. Section 4.3 describes how a constraint is enforced by constructing a method vine, the core of the SkyBlue algorithm. The algorithm described in these sections produces correct results, but its performance suffers as the constraint graph becomes very large. Section 5 presents several techniques used in the complete algorithm that significantly improve the efficiency of SkyBlue for large constraint graphs. More detailed information on SkyBlue, including complete pseudocode for the algorithm, is available in a technical report [7].

4.1 Adding Constraints

When a new constraint is added to the constraint graph it may be possible to alter the method graph to enforce it by selecting a method for the constraint, switching the selected methods of enforced constraints with the same or stronger strength, and possibly revoking one or more weaker constraints. This process is known as constructing a *method vine* or *mvine* (described in Section 4.3). **add-constraint** adds a new constraint **cn** to the constraint graph by performing the following steps:

1. Add **cn** to the constraint graph (unenforced) and try to enforce **cn** by constructing an mvine. If it is not possible to construct such an mvine, leave **cn** enenforced and return from **add-constraint**. In this case, the method graph is unchanged (it is still LGB).

2. Repeatedly try to enforce all of the unenforced constraints in the constraint graph by constructing mvines until none of the remaining unenforced constraints can be enforced. Note that each time an

[2] Reference [5] defines "locally-graph-better" such that directed cycles are prohibited. In this paper, the definition of LGB is modified so LGB method graphs may include directed cycles.

unenforced constraint is successfully enforced, one or more weaker constraints may be revoked. These newly-unenforced constraints must be added to the set of unenforced constraints.

3. Execute the selected methods in the method graph to satisfy the enforced constraints (as described in Section 2).

The second step must terminate because there are a finite number of constraints. Each time an unenforced constraint is enforced, one or more weaker constraints may be added to the set of unenforced constraints. These additional constraints may be enforcible, adding still weaker constraints to the set of unenforced constraints, but this process cannot go on indefinitely. Eventually the process will stop with a set of unenforcible constraints. When the second step terminates the method graph must be LGB, since no more mvines can be constructed.

As an example, suppose that **add-constraint** has just added $C2$ to the constraint graph and the current method graph is shown in Figure 2a. One way that an mvine could be constructed is by enforcing $C2$ with the method that outputs to $V2$ and revoking $C3$ (Figure 2b). Given this method graph, the second step would try constructing an mvine to enforce $C3$, possibly by revoking $C4$ (Figure 2c). At this point it is not possible to construct an mvine to enforce $C4$ so the second step terminates. This method graph is LGB. Alternatively, if the first mvine had been constructed by revoking $C1$ then the LGB method graph of Figure 2d would have been produced immediately and the second step would not have been able to enforce $C1$.

4.2 Removing Constraints

remove-constraint is very similar to **add-constraint**. When an enforced constraint is removed this may allow some unenforced constraints to be enforced, which leads to the same process of repeatedly constructing mvines. **remove-constraint** removes a constraint cn from the constraint graph by performing the following steps:

1. If cn is currently unenforced, remove it from the constraint graph and return from **remove-constraint**. Removing an unenforced constraint cannot make any other constraints enforcible so the method graph is still LGB.

2. Repeatedly try enforcing all of the unenforced constraints in the constraint graph by constructing mvines (adding revoked constraints to the set of unenforced constraints) until none of the unenforced constraints can be enforced. As in **add-constraint** this step eventually terminates with an LGB method graph.

3. Execute the selected methods in the method graph to satisfy the enforced constraints (as described in Section 2).

4.3 Constructing Method Vines

The SkyBlue algorithm is based on attempting to enforce an unenforced constraint by changing the selected methods of constraints with the same or stronger strength and/or revoking one or more constraints with weaker strengths. There are many ways this could be implemented, including trying all possible assignments of selected methods without method conflicts. The technique used in SkyBlue, known as constructing a method vine (or mvine), uses a backtracking depth-first search.

An mvine is constructed by selecting a method for the constraint we are trying to enforce (the root constraint). If this method has a method conflict with the selected methods of other enforced constraints, we select new methods for these other constraints. These new selected methods may conflict with yet other selected methods, and so on. This process extends through the method graph, building a "vine" of newly-chosen selected methods growing from the root constraint. This growth process may terminate in the following ways:

1. If a newly-selected method in the mvine outputs to variables that are not currently determined by any constraint, then this branch of the mvine is not extended any further.

2. If a newly-selected method in the mvine conflicts with a selected method whose constraint is weaker than the root constraint, then the weaker constraint is revoked, rather than attempting to find an alternative selected method for it. As a result, all of the methods in the mvine will belong to constraints with equal or stronger strengths than the root constraint.

3. If an alternative selected method is chosen for a constraint and there is a method conflict with another selected method in the mvine, then we cannot add this method to the mvine and must try another method. If all of the methods of this constraint conflict with other selected methods in the mvine, then the mvine construction process backtracks: previously-selected methods are removed from the mvine and the mvine is extended using other selected methods for these constraints. If no method can be chosen for the root constraint that allows a complete conflict-free mvine to be constructed, then the root constraint cannot be enforced.

Figure 3 presents an example demonstrating the process of constructing an mvine. A complete mvine is a connected subgraph of the method graph. An mvine is not necessarily a tree: separate branches may merge and it may contain directed cycles. If all of the constraint methods in the mvine have a single output, then an mvine will have the structure of a single stalk leading from the root constraint through a series of other constraints with changed selected methods. If there is a method with multiple outputs in the mvine, the mvine will divide into multiple branches with one branch for each output. The different branches cannot be extended independently since methods in different branches cannot output to the same variables. The backtracking search must take this into account by trying all possible combinations of selected methods for the constraints in the different branches.

5 Performance Techniques

The SkyBlue algorithm described in Section 4 works correctly, but its performance suffers as the constraint graph becomes very large. This happens because larger constraint graphs may contain greater numbers of unenforced constraints that SkyBlue has to try enforcing, and each attempt to construct an mvine may involve searching through more enforced constraints. The following subsections describe techniques used in SkyBlue to improve its performance with larger constraint graphs.

5.1 The Collection Strength Technique

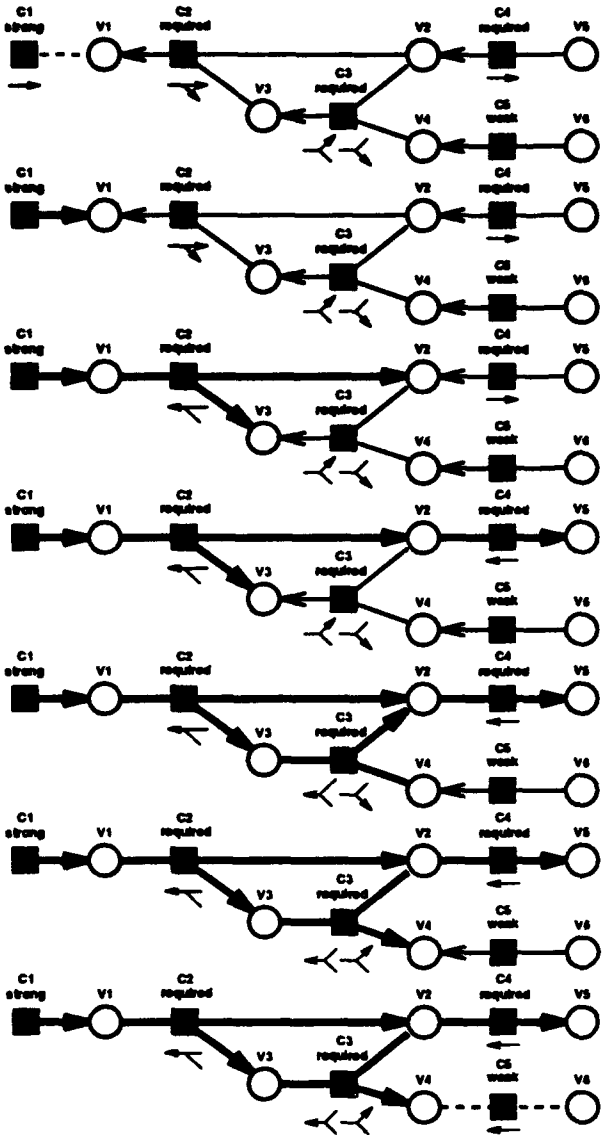
The initial SkyBlue method graph is empty and LGB. Every call to `add-constraint` or `remove-constraint` leaves an LGB method graph. Therefore, the current method graph must be LGB whenever `add-constraint` or `remove-constraint` is called. This fact can be used to avoid collecting and trying to enforce some of the unenforced constraints.

Whenever `add-constraint` adds a constraint `cn`, it is impossible for it to enforce any unenforced constraints with the same or stronger strength than `cn`, other than `cn` itself. If it was possible to enforce any such constraint after `cn` was added, then it would have been possible to enforce it before `cn` was added and the previous method graph would not have been LGB.

Whenever `remove-constraint` removes an enforced constraint `cn`, it is impossible to enforce any unenforced constraints that are stronger than `cn`. If it was possible to enforce any stronger constraint after `cn` was removed, then it would have been possible to enforce it before `cn` was removed and the previous method graph would not have been LGB. Note that unlike `add-constraint`, removing a constraint may allow unenforced constraints with the same strength to be enforced, as well as weaker ones.

5.2 The Local Collection Technique

If the method graph is LGB and a constraint is added or removed from the constraint graph, any unenforced constraints in a subgraph unconnected to the added or removed constraint clearly cannot be enforced. It is possible to be more selective: Whenever `add-constraint` is called to add a constraint `cn` and an mvine is successfully constructed to enforce it, it is sufficient to collect unenforced constraints that constrain variables downstream in the method graph from all of the "redetermined variables" whose determining constraint



Suppose we start with this method graph, and we want to enforce the **strong** constraint $C1$ by building an mvine.

First, $C1$'s selected method is set to its only method so it determines $V1$.

This causes a method conflict with $C2$ so we have to enforce $C2$ with its other method.

This causes method conflicts with $C3$ and $C4$. Suppose we process $C4$ first: we can simply switch its selected method so it determines $V5$. $V5$ is not determined by any other constraints so we don't have to extend this branch of the mvine.

We have to process $C3$ by choosing another method. Suppose we try the one that determines $V2$. This is not permitted because it causes a method conflict with $C2$, which is already in the mvine.

Therefore, we have to backtrack and try another method for $C3$. Suppose we now try the method that determines $V4$ (causing a method conflict with $C5$).

Now we need to handle $C5$. Because it is weaker than $C1$ we don't have to find an alternative method but can simply revoke it, producing this final method graph.

Figure 3: Constructing an mvine. Methods in the mvine are drawn with thicker lines.

has changed. Whenever `remove-constraint` is called to remove a constraint `cn`, it is sufficient to collect unenforced constraints that constrain variables downstream from the variables previously determined by `cn`.

Whenever SkyBlue successfully constructs an mvine, additional unenforced constraints can be added to the set of collected unenforcec constraints by scanning downstream from the newly-redetermined variables. As each of these constraints is processed (it is enforced, or it is determined that it cannot be enforced) it can be removed from the set. When the set is empty there are no more unenforced constraints that can be enforced.

A similar technique can be used to reduce the number of methods executed. Rather than executing the selected methods of all enforced constraints in the constraint graph, it is only necessary to collect and execute the selected methods of newly-enforced constraints, and methods downstream of redetermined variables.

5.3 Walkabout Strengths

An mvine is constructed by repeatedly choosing a new selected method for a constraint and then trying to extend the mvine from the outputs of this method. It will be possible to complete the mvine below these outputs only if the mvine eventually encounters undetermined variables or constraints weaker than the root constraint, and there are no method conflicts between different branches of the mvine. If SkyBlue could predict that one of these conditions was untrue then the selected method could be rejected immediately without trying to extend the mvine.

The DeltaBlue algorithm predicts whether a constraint can be enforced by using the concept of ***walkabout strengths*** [4]. A variable's walkabout strength is the strength of the weakest constraint that would have to be revoked to allow that variable to be determined by a new constraint. This could be the strength of the constraint that currently determines the variable or the strength of a weaker constraint elsewhere in the method graph that could be revoked after switching the selected methods of other constraints. If the variable is not currently determined by any constraint then the walkabout strength is defined as **weakest**, which is a special strength weaker than any constraint. A variable will also have a walkabout strength of **weakest** if it can be left undetermined by switching selected methods without revoking any constraints.[3]

One important property of DeltaBlue's walkabout strengths is that they can be calculated using local information. The walkabout strength of a variable determined by a constraint can be calculated from the constraint's strength, its methods, and the walkabout strengths of the rest of the constraint's variables. If the method graph has no cycles (required for DeltaBlue), all of the variable walkabout strengths can be updated by setting the walkabout strengths of all undetermined variables to **weakest** and processing each enforced constraint in topological order to set the walkabout strengths of the determined variables.

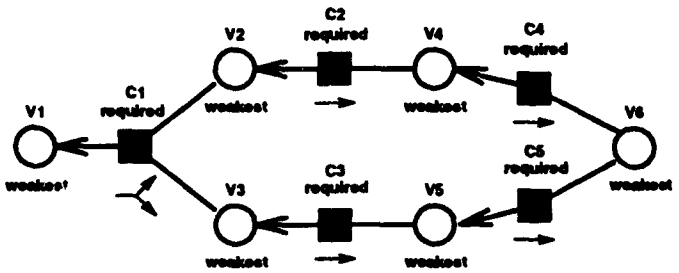


Figure 4: Method graph with a possible conflict.

There is a problem with using walkabout strengths in SkyBlue because methods may have multiple outputs. Consider the method graph of Figure 4. DeltaBlue would correctly calculate the walkabout strengths of $V2$–$V6$ to be **weakest**. But what about $V1$? The walkabout strengths of $V2$ and $V3$ imply that $V1$ should have a walkabout strength of **weakest**, since the alternative (multi-output) method for $C1$ can be chosen that outputs to $V2$ and $V3$, which both have **weakest** walkabout strengths. However, it is not possible for a method to set *both* $V2$ and $V3$ simultaneously, without revoking one of the **required** constraints. Simply switching methods would lead to a method conflict with both $C4$ and $C5$ determining $V6$. However, this cannot be detected without exploring the graph, which would remove one of the benefits of walkabout strengths (i.e., they can be calculated using local information).

In SkyBlue, the definition of walkabout strength is modified. A variable's walkabout strength is defined as a ***lower bound*** on the strength of the weakest constraint in the current method graph that would need to be revoked to allow the variable to be determined by a new constraint. SkyBlue uses the modified definition of walkabout strengths to reject methods when constructing an mvine: if any of the outputs of a method have walkabout strengths equal to or stronger than the root constraint, then it is not possible to complete the mvine using this method. The use of walkabout strengths cannot eliminate all of the backtracking during mvine construction but it can reduce it considerably.

Whenever SkyBlue successfully constructs an mvine it modifies the method graph, so the walkabout strengths must be updated to correspond to the new method graph. This is done by processing all of the

[3] Another interpretation of the **weakest** strength is that each variable has an implicit stay constraint with a strength of **weakest**, which specifies that the variable value doesn't change unless a stronger constraint determines it.

enforced constraints in the constraint graph (in topological order) and recalculating the walkabout strengths of the determined variables. It is possible to apply the technique from Section 5.2 in this situation by processing only the enforced constraints downstream of the redetermined variables.

SkyBlue uses the modified definition of walkabout strengths to simplify the processing of cycles. If the method graph contains directed cycles, it is not possible find a topological sort for the constraints. The walkabout strengths for variables in the cycle could be calculated by examining all of the constraints in the cycle, but this would require non-local computation. Instead, SkyBlue chooses a selected method in the cycle and calculates the walkabout strengths of its outputs as if all of its input variables in the cycle had walkabout strengths of `weakest`. This is guaranteed to be a correct lower bound. This simplifies the updating of walkabout strengths at the cost of increasing the search when constructing an mvine, because the walkabout strengths in a cycle and downstream may be weaker than necessary.

5.4 Comparing Performance Techniques

For regression testing and performance tuning, a random sequence of 10000 calls to `add-constraint` and `remove-constraint` was generated and saved. Using this sequence it is possible to measure how the performance of SkyBlue is improved by the performance techniques described above.

local collection	*walkabout strengths*	*time (seconds)*	*number mvines attempted*	*number mvines constructed*	*number backtracks*
on	on	25.3	24222	5840	12748
on	off	47.2	24222	5840	196012
off	on	44.1	77354	5826	36262
off	off	125.5	77354	5826	571534

Figure 5: Measurements collected while executing a sequence of 10000 constraint operations in SkyBlue with different performance techniques enabled.

Figure 5 shows the timing results when executing the sequence with four different configurations of the SkyBlue algorithm. The *local collection* column specifies whether the technique from Section 5.2 was used to collect constraints local to the added or removed constraint when enforcing and executing methods, versus processing all of the constraints in the constraint graph. The *walkabout strengths* column specifies whether variable walkabout strengths were used to improve mvine searches and updated whenever an mvine was constructed, as described in Section 5.3. The times in the third column show that SkyBlue is most efficient with both techniques enabled, about half the speed with either technique disabled, and exceedingly slow with neither of the techniques enabled. The collection strength technique of Section 5.1 was enabled in all four cases. Disabling this technique did not change the times as much as the other two techniques.

These timings are explained by the remaining three columns, which record the number of times SkyBlue attempted to enforce a constraint by constructing an mvine, the number of times the mvine was successfully constructed, and the number of times backtracking occurred while trying to construct an mvine. These timings can be interpreted as follows: the local collection technique saves time by reducing the number of attempts to construct mvines and the number of constraint methods executed. The walkabout strength technique saves time by reducing the amount of backtracking when constructing an mvine. This particularly reduces backtracking (and time) when there are numerous unsuccessful mvines, such as when the local collection technique is disabled.

6 SkyBlue Applications

SkyBlue is currently being used as the constraint solver in Multi-Garnet [8], a package that extends the Garnet user interface construction system [6] with support for hierarchies of multi-way constraints. SkyBlue is also currently being used as the constraint solver in an implementation of the Kaleidoscope language [3] and as an equation manipulation tool in the Pika simulation system [1].

6.1 Multi-Garnet

Garnet is a widely-used user interface toolkit built on Common Lisp and X windows [6]. However, Garnet only supports one-way constraints, all of which must be required (no hierarchies). The Multi-Garnet package uses the SkyBlue solver to add support for multi-way constraints and constraint hierarchies to Garnet [8].



Figure 6: Three views of a scatterplot built within Multi-Garnet: The initial scatterplot, the initial scatterplot after moving the X-axis, and the initial scatterplot after scaling the point cloud by moving a point.

Figure 6 shows three views of a graphic user interface constructed in Multi-Garnet: a scatterplot displaying a set of points. SkyBlue constraints are used to specify the relationship between the screen position of each point, the corresponding data value, and the positions and range numbers of the X and Y-axes. As the scatterplot points and axes are moved with the mouse, SkyBlue maintains the constraints so that the graph continues to display the same data.

The scatterplot application exploits many of the features of SkyBlue. SkyBlue resatisfies the constraints quickly enough to allow continuous interaction. The different interactions (move axis, move point cloud, scale point cloud, etc.) are defined by adding weak stay constraints to specify variables that should not be changed. Multi-way constraints allow any of the scatterplot points to be selected and moved, changing the axes data. This changes the positions of the other points, reshaping or moving the point cloud. Finally, the scatterplot uses constraints with multi-output methods, such as a constraint with three two-output methods that maintains the relationship between the X-coordinates of the ends of the X-axis, the range numbers displayed at the ends of the axis, and the scale and offset variables used to position points relative to the axis. It would be difficult to build this application in Garnet without maintaining some of the relationships using other mechanisms in addition to the Garnet constraint solver.

6.2 The Pika Simulation System

SkyBlue is being used as an equation manipulation tool in a version of the Pika simulation system [1]. Pika constructs simulations in domains such as electronics or thermodynamics by collecting algebraic and differential equations representing relationships between object attributes. For example, in a simulation of an electronic circuit, one equation would relate the voltage across and the current through a particular resister. Pika processes these equations and passes them to a numerical integrator that calculates how the object attributes change over time.

Pika uses SkyBlue to manipulate the collected equations. Each equation is expressed as a SkyBlue constraint with one method for each possible output variable. SkyBlue chooses one method from each constraint so that no two constraints select the same output variable, and topologically orders the selected methods. Pika uses the ordered list of selected methods to set up the numerical integrator. Note that Pika does not use SkyBlue to maintain the constraints (equations) directly, but rather uses it to process the equations for the numerical integrator, which will maintain the equations during the simulation.

During equation processing, Pika takes advantage of SkyBlue's support for constraint hierarchies to influence the methods selected. There may be many possible ways to directionalize a given set of equations, leaving different sets of variables constant. Within the simulation, it may be preferable to keep some variables constant over others. This is represented by adding weak stay constraints to variables that should remain constant. SkyBlue will choose an equation ordering that leaves these variables constant, if possible.

Pika uses SkyBlue's facilities for incrementally adding and removing constraints to update the sorted list of equation methods as equations are added and removed. This may occur while the simulation is executing. For example, when the temperature of a container of water increases and it starts to boil, a different set of equations describing its behavior is activated.

Often there are cycles in the sets of selected methods produced by SkyBlue. Currently, Pika handles these cycles by extracting the equations in the cycle, and passing them to an symbolic mathematics system which tries to transform them to a non-cyclic set of equations. Pika replaces the cycle of equations by the reduced equations, SkyBlue (incrementally) updates the constraint graph, and Pika processes the new ordered list of selected methods.

Acknowledgements

Thanks to Alan Borning, Ralph Hill, and Brad Vander Zanden for useful discussions and comments on earlier versions of this paper. This work was supported in part by the National Science Foundation under Grants IRI-9102938 and CCR-9107395.

References

[1] Franz G. Amador, Adam Finkelstein, and Daniel S. Weld. Real-Time Self-Explanatory Simulation. In *Proceedings of the National Conference on Artificial Intelligence*, 1993. To appear.

[2] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint Hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[3] Bjorn Freeman-Benson and Alan Borning. The Design and Implementation of Kaleidoscope'90, A Constraint Imperative Programming Language. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 174–180, April 1992.

[4] Bjorn Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[5] John Maloney. *Using Constraints for User Interface Construction.* PhD thesis, Department of Computer Science and Engineering, University of Washington, August 1991. Published as Department of Computer Science and Engineering Technical Report 91-08-12.

[6] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, and Ed Pervin. Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment. *IEEE Computer*, 23(11):71–85, November 1990.

[7] Michael Sannella. The SkyBlue Constraint Solver. Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington, February 1993.

[8] Michael Sannella and Alan Borning. Multi-Garnet: Integrating Multi-Way Constraints with Garnet. Technical Report 92-07-01, Department of Computer Science and Engineering, University of Washington, September 1992.

[9] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software—Practice and Experience*, 1992. In press.

A Real Time Extension to Logic Programming Based on the Concurrent Constraint Logic Programming Paradigm

Tony Savor, Paul Dasiewicz
(tsavor@vlsi.uwaterloo.ca, dasiewic@vlsi.uwaterloo.ca)

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Abstract

Although concurrent logic programming languages provide a suitable implementation environment for real time systems, they fail to give any notion of temporal correctness. We define a set of semantics whereby temporal constraints, consisting of delay, maximum execution time, and priority specifications can be implemented into existing concurrent logic programming languages. The notion of temporal inheritance is described which allows distribution of common temporal constraints to processes that are all part of the same task. We give some examples illustrating the utility of the language extension.

1 Introduction

Real time systems (RTS) can be defined as systems whose result not only depends on the correctness of the output, but also the *time* at which the output is produced. *Hard* RTSs have deadlines which if missed can be either disastrous (missile guidance systems) or useless (stock market prediction). *Soft* RTSs such as telephone exchanges, on the other hand can tolerate a certain proportion of missed deadlines with little or no catastrophic effect.

Logic programming languages were found to be suitable for the implementation of real time control systems [1, 6, 8]. Declarative nature of programming, automated translation of software design specifications to logic [5, 7, 11], and reduced development time [1] are some of the advantages offered by these languages.

Currently, there exist two attempts to extending concurrent logic languages for real time. Fleng [8], incorporates a *delay* predicate which binds a logical variable after a specified amount of time. PARLOG-RT [6] introduces predicates, *after*, *before*, and *delay*. The *before* and *after* predicates allow exception handling if deadlines are preceded or missed respectively, while the delay predicate executes a specified goal after a finite duration of time.

The problem with the current approaches is the lack of process urgency specification and scheduling. Under heavy system loads multiple jobs contending for CPU time will be serviced in an order based on some non-temporal algorithm. The result is that lower urgency jobs may execute before higher urgency ones possibly causing system failure.

The objective of this work is twofold. First, to provide a set of formal semantics by which *existing* concurrent logic programming languages can be extended for real time and second, to allow schedulability of the language by assigning urgencies to all processes of a program.

1.1 Concurrent Constraint Logic Programming

The work presented here is centered around the notion of concurrent constraint logic programming(cclp) [9]. We now give a brief introduction to the relevant issues of cclp.

The architecture of a cclp language, shown in Figure 1, consists of an executing program communicating with a central store. A program is composed of a number of agents,[1] each of which communicates

[1] Here we use the term agent to refer to a process in the constraint logic programming scheme.

independently with the store via ask and tell operations (corresponding roughly to read and write).

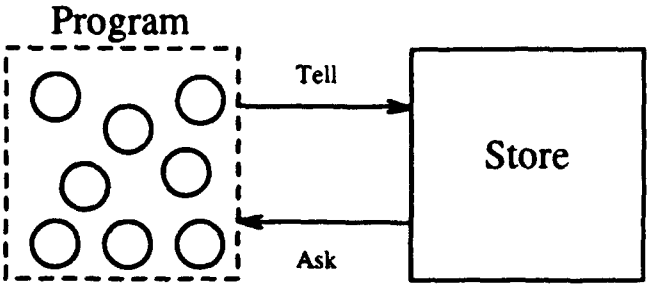


Figure 1: Architecture of a cclp Language

1.1.1 Store

The store is best described as being analogous to a memory map of a traditional computer. The addresses of a memory map corresponds to variables in a store (ie a variable is mapped to a given address) and the data elements correspond to constraints which are *told* on the variables by agents. The ask and tell operations differ from a memory map in that they read and post *constraints* rather than data values.

Constraints, which remain in the store for the lifetime of the computation, are added to the existing constraints of a variable in a conjunctive fashion. The store can be considered a fundamentally parallel model of memory since agents telling constraints will not overwrite previous entries and all entries will become part of the final solution.

Variables in the store and their associated constraints must be defined under a specific domain of discourse which we shall assume to be the set of real numbers for purposes of this paper. The result of a successful computation is a set of variables and their associated constraints which when solved by an appropriate constraint solver[2] yield a general solution.

A computation is said to *fail* if there exists a variable in the store, whose associated conjunction of constraints entails the empty set. Such a variable is called an *inconsistent* variable and a store containing an inconsistent variable is correspondingly called an inconsistent store.

1.1.2 Tell

A tell operation consists of adding a conjunctive constraint to the existing constraints for a given variable. Since the conjunctions, $c_0 \wedge c_1$ and $c_1 \wedge c_0$ are equivalent, it should be apparent that the order of which constraints are posted to the store is irrelevant. Thus the store has the notion of *stability*, or more precisely, a store where constraint c_0 is posted before c_1 is equivalent to a store in which constraint c_1 is posted before c_0.

For computations to remain consistent, each variable must be checked for consistency *before* a tell is allowed to succeed. Thus a tell operation must block the telling agent until it is determined that the constraint which is to be posted to the store is consistent. We call this type of tell *atomic tell.* If the constraint to be posted by atomic tell is inconsistent with the store, the tell operation fails.

1.1.3 Ask

An ask operation checks if a specified variable in the store entails the asked constraint. To maintain the notion of stability, an asked constraint in which there does not exist enough information in the store to ensure that this constraint will me met after all future tell operations by other agents will suspend. If a constraint is posted by another agent such that the store entails the asked constraint, the previously suspended agent is allowed to proceed. Conversely if the constraints posted falsify the asked constraint, the ask operation fails.

[2]The constraint solver will depend on the domain of discourse.

The notions of stability within the store imply that a successful ask operation will be successful if it is asked any time in the future provided that the store remain consistent. Thus we have a framework by which ask operations will suspend waiting for tell operations, providing an automatic computation synchronisation mechanism allowing all agents to execute concurrently.

1.1.4 Eventual Tell

Atomic tell is a computationally expensive operation since a consistency check must be done before the operation is allowed to succeed. If one could guarantee that *all* posted constraints will be consistent with the store then this consistency check could be omitted. This type of tell operation is called *eventual tell*[3].

An eventual tell always succeeds and telling agents do not suspend waiting for constraints to reach the store. The constraints are put into a repository and eventually migrated to the store. Given that the store is stable, agents asking constraints not yet entailed by the store will suspend until the store entails the asked constraints. Thus we have a framework of asking agents' waiting for telling agents' constraints to be posted to the store and ensuring *correct* computation regardless of the time constraints take to migrate to the store through the repository.

2 Incorporation of Temporal Constraints

We now augment the cclp framework as to introduce the notion of time.

2.1 Temporal Constraints

The proposed structure of the real time cclp language (rtcclp) is given in Figure 2. Agents post constraints to the buffer and proceed with computation (ie eventual tell). Agents have no notion of how long it will take to post a constraint told by an eventual tell to the store. By making the posting delay of eventual tell specifiable, we can introduce the notion of time into the computation. The buffer will serve as a scheduler since it will effectively suspend computation until constraints are posted. We call this buffer a *temporal buffer*.

Agents telling constraints through the temporal buffer must have some way of synchronising with the constraints when posted, or more precisely should suspend until their constraint is posted to the store by the temporal buffer. We thus introduce the notion of a *temporal tell*. The temporal t a compound operation composed of an eventual tell through the temporal buffer followed by an ask of ld constraint from the store. The temporal tell effectively communicates scheduling information to the temporal buffer and suspends until scheduled by having its told constraint posted. Since eventual tells must be consistent with the store, all temporal constraints must be consistent with previously posted constraints. This issue shall be readdressed when temporal constraints are defined in section 2.3.

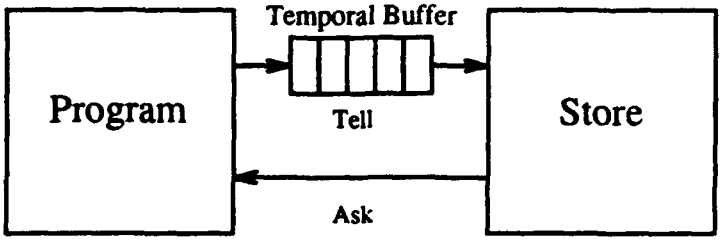


Figure 2: Architecture of a Real-Time cclp Language

2.2 Real-Time Constraints

Specification of start and finish times are the two parts of a temporal constraint. An infinite or zero specification of finish or start time respectively is used to indicate a non-specification. For example an agent

[3]The motivation of eventual tell is largely distributed systems. The interested reader is referred to [9] for details.

that wishes to be delayed for a period of *Delay* but not have a definite finish time could post a constraint such as given in equation 1 (where T_c is the current time, t_s the start time and t_f is the agents desired finish time). An agent with a specific deadline by which computation must terminate would post a temporal constraint similar to equation 2. A *window* of execution can be established for an agent by specifying both the *delay* and *maximum execution time* as finite, non-zero values. From these examples it should be apparent that an agents complete temporal properties can be expressed by specifying only the parameters, *Delay* and *MaxExecutionTime*.

$$t_s \geq (T_c + Delay) \wedge t_f \leq (T_c + \infty) \tag{1}$$

$$t_s \geq (T_c + 0) \wedge t_f \leq (T_c + Max\ Execution\ Time) \tag{2}$$

2.3 Temporal Tell

We now formalise the notion of a temporal tell. A temporal tell consists of telling the start and finish time constraints of the currently executing agent to the temporal buffer followed by an ask of the start constraint from the store. Since the temporal buffer is a global entity, it performs scheduling based on agents start and finish times. When an agent is to be scheduled for computation, the temporal buffer posts a constraint of the form $t_s >= T_c$, satisfying the agents ask and allowing computation to proceed[4]. Since T_c (current time) is ever increasing, and an agent will suspend until its asked inequality is satisfied (ie will not perform any tells while suspended), it should be apparent that all temporal tells will be consistent.

The reader should note that each agent's start and finish times as posted by the temporal tell operation are entirely consumed by the temporal buffer. The start time places a *lower bound* on the time at which the agent will be unsuspended, while the finish time indicates to the temporal buffer the agent's *urgency* of execution. This notion of agent urgency becomes important when more than one agent wish to be executed at any point in time. We shall re-address the specification of agent urgency when priorities are introduced into this framework in section 3.3.

The temporal tell mechanism requires that each agent be allocated a unique variable, (t_s) in the store to which all temporal constraints are told. We call this variable a *temporal variable* and assume each agent's temporal variable to exist for the lifetime of the agent. At any given point in time, each temporal variable entails its associated agent's start time.

To allow newly created agents with higher urgencies to preempt the currently executing one, processes will be required to execute asks of the form $t_s >= T_c$ on a regular basis. This gives the temporal buffer the ability to execute agents for a finite period of time, *extime*, by posting a constraint, $t_s >= T_c + extime$.

3 Incorporating Temporal Tells into a Concurrent Logic Programming Language

We now turn our attention to the semantics associated with incorporating temporal tells into a concurrent logic programming language. Although the concepts presented here should be widely applicable throughout the family of commit choice languages, our implementation was centered around PARLOG [2].

It should be noted that although the extensions proposed are rooted in concurrent constraint logic programming, one need not implement a store to realize rtcclp.

3.1 Compound Guard

The clause syntax for rtcclp is as follows:

$$Head \leftarrow Guard|TemporalGuard|Body.$$

The guard of concurrent logic programming languages is augmented with a temporal guard to form a *compound guard*. The two commit operators (|) imply that the *Guard* must succeed before the *Temporal Guard*

[4] We assume the temporal buffer to have knowledge of when agents terminate so as to be able to schedule another agent.

is executed and the *Temporal Guard* must commit before the body of the clause is executed. The temporal guard performs a temporal tell and commits upon its ask constraint being satisfied by the store. Since both the *Guard* and *TemporalGuard* are optional, we use the commit operator notation below to specify clauses without a *Guard*, *TemporalGuard* and compound guard respectively.

$$Head \leftarrow |TemporalGuard|Body.$$

$$Head \leftarrow Guard||Body.$$

$$Head \leftarrow ||Body.$$

The temporal guard consists of specification of an agent's *Delay* and $MaxExecutionTime$ as shown below.

$$Head \leftarrow Guard|Delay, MaxExecutionTime|Body.$$

The temporal buffer ensures that an agent's asked constraint will be posted *after* $(T_c + Delay)$ seconds, and early enough so that its finish time will be less than $T_c + MaxExecutionTime$ (where T_c is sampled after the delay portion of the temporal guard has succeeded).

The *Delay* parameter of the temporal guard may be an input parameter of the clause head, however if uninstantiated at time of temporal guard evaluation, the agent will suspend until the variable becomes instantiated, after which the agent will execute a temporal tell (effectively sampling T_c *after Delay* gets instantiated). The $MaxExecutionTime$ specification is to be specified as a constant at compile time. This will ensure, that at runtime, deadline specifications are not unspecified, leading to possible system error. The situation when a fixed specification time becomes awkward is with common procedures which are called by multiple agents with different finish times. The next section introduces a set of semantics to circumvent this problem.

3.2 Inheritance of Temporal Specifications

To permit multiple clauses to share identical temporal specifications (start and finish times) as a given parent clause we introduce the notion of *temporal constraint inheritance*.

Consider the abstract program below. Clause A contains a temporal guard which specifies a finite execution time. Subgoal $B1$ will unify with the head of clause $B1$. Clause $B1$ will *inherit* the finish time of clause A. Similarly clause $B4$ will unify with clause $B1$'s subgoal $B4$ and it too will inherit the finish time of clause A. Inheritance can be conceptualized as having all child process inherit the finish time of their parent. All processes which inherit a finish time from a common parent will be referred to as a *task*. Temporal guards serve to *modify* the finish time of a process, in effect creating a new task. It should be noted that clauses inherit the actual *finish time*, $(T_c + MaxExecutionTime$, where T_c is sampled by the parent), not just $MaxExecutionTime$.

$$\leftarrow A.$$

$$A \leftarrow |0, MaxExecutionTime_1|B1, B2, B3.$$

$$B1 \leftarrow ||B4, B5, B6.$$

$$B4 \leftarrow ||B7, B8.$$

$$B8 \leftarrow |TemporalGuard_2|B9.$$

The notion of tasking creates non-determinism in the finish time if (for example) clause B8's temporal guard has a long finish time, $(T_c2 + MaxExecutionTime2)$ and clause B1's temporal guard has a short finish time, $(T_c1 + MaxExecutionTime1)$, such that $(T_c1 + MaxExecutionTime1) < (T_c2 + MaxExecutionTime2)$, the specification of $MaxExecutionTime1$ may not be met due to the fact that it could be waiting for the temporal guard of clause B8. This requires all subclauses which are part of the same functional task to *inherit* their temporal specifications from a common parent. A programmer specified temporal guard creates a new task and as such parent tasks should not be dependent on child tasks, but remain functionally and temporally independent.

Semantically, clauses inherit start times as well as finish times, however since start times have been met given that the temporal guard has committed, they will also be met after the start times are inherited given the fact that T_c is sampled by the parent. For this reason clauses need only inherit finish times.

3.3 Priorities

Programmers often require more temporal expressive power than can be bought with the temporal specifications described. For example, supervisory processes in a system are usually required to be of higher urgency than worker process. For this reason, we introduce the notion of priorities.

Syntactically, a clause priority is specified as an optional third element in the temporal guard and does not change any of the commitment semantics. Like $MaxExecutionTime$, priority information is entirely consumed by the temporal buffer and is treated as scheduling information. In the event of an unspecified priority, the system defaults to some predetermined middle priority, giving the programmer the flexibility of using priorities higher or lower than the default.

Inheritance of priorities is identical to that of finish times. Processes that execute a temporal guard have their priority modified to reflect that of the temporal guard. Child processes that are spawned by a parent *inherit* the finish time and priority of the parent. In the example below, processes, B, C, D and E execute at $MaxExecutionTime_1$ and $Priority_1$ while processes F and G execute at $MaxExecutionTime_2$ and at the system default priority.

$$\leftarrow A$$

$$A \leftarrow |Delay_1, MaxExecutionTime_1, Priority_1|B, C.$$

$$B \leftarrow ||D, E.$$

$$D \leftarrow |Delay_2, MaxExecutionTime_2|F, G.$$

The inclusion of priorities into the framework should be regarded solely as *process urgency information.* Should other urgency information, such as process execution time or confidence intervals of execution, be available, it could be incorporated into this framework in an identical fashion.

3.4 Input & Output

Program input and output (I/O) such as handshaking introduces non-determinism into deadline specifications since a process belonging to a task may wait indefinitely for an external device over which it has no control. It becomes impossible to bind deadlines around such events. For this reason we provide a set of semantics for non-deterministic I/O by which the programmer can effectively perform I/O and maintain the temporal properties of tasks.

We begin by separating possibly non-deterministic I/O predicates from the remainder of the predicates of the language. Each I/O predicate will *implicitly* execute a temporal guard *before* the actual I/O predicate is executed. The effect is that each I/O predicate creates a separate task for itself and does not inherit its parents finish time. The problem with this scheme is that all predicates following an I/O predicate would then inherit its finish time. For this reason we restrict the placement of I/O to the *final* predicates of the guard as shown below.

$$A \leftarrow |TemporalGuard_1|B_1, B_2, B_3, \ldots, B_i.$$

$$B_1 \leftarrow G_1, G_2, \ldots, G_j \;\&\; I/O_1, I/O_2 \ldots I/O_k|TemporalGuard_2|Body_2.$$

The predicates $G_1 \ldots G_j$ of the second clause form the standard component of the concurrent logic programming guard. The *&* is a sequential operator[5] which ensures that the standard guard completes execution *before* the I/O is begun. After all the I/O goals have completed, the temporal guard is executed and creates a new task for subgoals in the body of the clause.

The net effect is that that the parent clause's task is responsible for *scheduling* the I/O operations within the duration of its finish time. In the example above, the first clause's subgoal in the body, B_1 unifies with the head of the second clause. Guard predicates $G_1 \ldots G_j$ are executed under the first clause's task. Upon reaching the I/O predicates, the task of the first clause is terminated and each I/O predicate executes its temporal guard and creates its own task. When all I/O predicates have succeeded, the guard commits which invokes the clause's temporal guard, effectively creating a new task for the clause's body.

[5] We borrow this sequential operator notation from PARLOG [2].

The temporal guards for the I/O subgoals are specified when the language is implemented and are not user modifiable. Typically I/O predicates are given a high system priority, infinite $MaxExecutionTime$ and a delay time of zero.

4 Examples

We now turn our attention to the implementation of a few examples which illustrate the utility of the extensions.

4.1 Delay

A finite delay can be modeled as a delayed start time with an unspecified finish time.

$$\leftarrow A.$$

$$A \leftarrow |Delay, \infty|Body.$$

Above, when the goal unifies with the clause, the temporal guard waits at least $Delay$ before committing.

4.2 Timeout

We can also model an action that waits for an input for a fixed amount of time and if the input does not occur, a timeout handler is executed. In the example below, $ProcessInput$ is to be executed should the input occur before $Waittime$ and $ProcessTimeout$ if the timeout expires. Neither of the clauses uses a $MaxExecutionTime$ specification and default to the system priority.

$$\leftarrow A.$$

$$A \leftarrow Input(x)|0, \infty|ProcessInput.$$

$$A \leftarrow |Waittime, \infty|ProcessTimeout.$$

4.3 Periodic Processes

A periodic process such as the sampling of a computer keyboard is a task which is performed repetitively, say every D units of time. The example below highlights the design of such a process. $Scan$ is invoked by the goal and delays D time units before committing. After commitment, $Scan$ and $Key_Pressed$ predicates execute in parallel.

$$\leftarrow Scan.$$

$$Scan \leftarrow |D, \epsilon|Scan, Key_Pressed.$$

$$Key_Pressed \leftarrow YES|0, \delta|Process_Key.$$

$$Key_Pressed \leftarrow NO||.$$

Both $Scan$ and $Key_Pressed$ have maximum execution times of ϵ as defined by the temporal guard. It should be apparent that the $Scan$ clause will be re-invoked on the interval $[0, \epsilon]$ time units after unsuspension of the temporal guard. This mechanism allows the programmer to effectively specify the worst case error in sampling due to program execution and take the factor into account in the calculation of D.

The $Key_Pressed$ clauses process the input depending if a key was pressed or not. In the key pressed case, a new task is created with a $MaxExecutionTime = \delta$. For this example, it is required that a key be processed before the next sampling. Based on this knowledge, we calculate the worst case execution time for δ. Initially, the temporal guard of the $Scan$ clause is executed at, t_0. In the worst case, $Scan$ will re-invoke itself at time t_0 and the temporal guard for the $Key_Pressed$ clause will be executed at time $t_0 + \epsilon$ (assuming a key was pressed). The $Process_Key$ must be executed *before* the next scan interval, $(t_0 + D)$, and thus $(t_0 + \epsilon + \delta < t_0 + D)$, or $(\epsilon + \delta < D)$.

With little or no modification, this segment of code could serve as an interrupt generator and handler for a larger system. By specifying high system priorities, one could ensure that these clauses preempt lower priority system processes and hence would function very similarly to a hardware generated interrupt.

4.4 Representation of State Machines

The state machine of Figure 3 can be represented in rtcclp as shown below.

$$\leftarrow A.$$

$$A \leftarrow X|TemporalGuard_1|B.$$

$$B \leftarrow Y|TemporalGuard_2|C.$$

$$C \leftarrow Z|TemporalGuard_3|A.$$

The inputs of the state machine (X, Y and Z) treated as non-deterministic I/O and are placed in the guards. The temporal guard specifies the processing time of transitions *after* the input has occurred. It is interesting to note that each task is responsible for scheduling the next. For example if the state machine is currently in state A and an X transition is received, $TemporalGuard_1$ is executed which forms a new task. The newly created task is completed when subgoal B succeeds or more precisely when the state machine is in state B waiting for input Y.

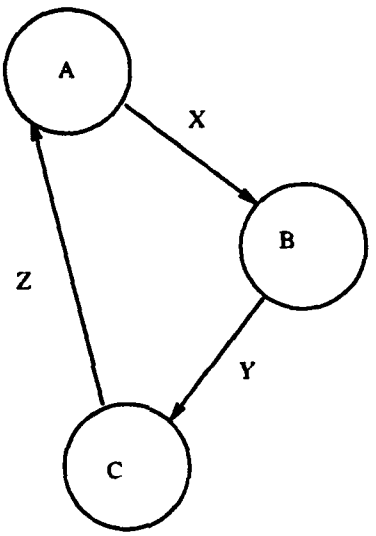


Figure 3: Typical Finite State Machine

5 Conclusions

We have defined a methodology by which to incorporate real time constraints into a concurrent logic programming language based on the concurrent constraint logic programming paradigm. Time is introduced into the logical framework by making the time taken to post constraints deterministic. The rtcclp language was implemented by modifying the abstract machine of PARLOG [3, 4]. The expressive power of rtcclp was found to be natural and effective in the implementation of a small private branch telephone exchange (PBX). At the programmer level, tasks and inheritance allow for increased programmer expressiveness, while at the emulator level, tasks are used to group work. Since the processor executes a single task with the highest urgency[6], tasks are completed sequentially rather than attempting to service many processes at once by timeslicing.

Preliminary simulations indicate approximately 10% decrease in performance of the real time extended PARLOG based on the PARLOG benchmarks described in [3]. Simulations of the PBX indicate almost two orders of magnitude decrease in CPU utilization between the real time and non-real time PARLOG due to the ability to suspend tasks for a finite delay. As expected, real time PARLOG produced results which had

[6] Urgency is a function of the current time, task priority, task deadline and scheduling algorithm.

greater probabilities of being within temporal specification than standard PARLOG. Further details of the implementation of real time PARLOG can be found in [10].

Acknowledgements

The authors wish to thank Keith Clark of Imperial College, London for providing documentation and source code for the PARLOG language and emulator, without which verification of the ideas presented would not have been possible. Gratitude is also expressed to Douglas Renaux of the University of Waterloo for his useful comments and provision of the PARLOG private branch exchange software.

References

[1] J.L. Armstrong, N.A. Elshiewy, and R. Virding. The phoning philosopher's problem or logic programming for telecommunications applications. In *Proceedings of the 3rd IEEE Symposium on Logic Programming, Salt Lake City*, pages 28–33, 1986.

[2] Keith Clark and Steve Gregory. PARLOG: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[3] Jim A. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors.* PhD thesis, Dept. of Computing, Imperial College, London, 1988.

[4] Jim A. Crammond. The abstract machine and implementation of parallel PARLOG. Technical report, Dept. of Computing, Imperial College, London, July 1990.

[5] Andrew Davison. From Petri Nets to PARLOG. Technical Report PAR 91/6, The PARLOG Group, Imperial College, London, March 1990.

[6] N.A. Elshiewy. Logic programming for real-time control of telecommunication switching systems. *Journal of Logic Programming*, pages 121–144, August 1990.

[7] David Gilbert. Implementing LOTOS in PARLOG. Technical Report PAR 87/1, The PARLOG Group, Imperial College, London, January 1987.

[8] Martin Nilsson. Mobile robot control with concurrent logic languages. In *Proceedings Euromicro Workshop on Real Time*, pages 42–46, 1990.

[9] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, 1989.

[10] T. Savor and P. Dasiewicz. Extending PARLOG for real-time. to appear.

[11] T. Taguchi and H. Hasegawa. Implementation of SDL switching systems using a parallel logic programming language. In *SDL '91: Evolving Methods*, pages 407–420, 1991.

Synthesis of Constraint Algorithms*

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304
smith@kestrel.edu

1 Introduction

Constraint propagation is one of the key operations on constraints in Constraint Programming. In a constraint program, a constraint set partially characterizes objects of interest and their relationships. As committments are made that further characterize some object, we want to infer consequences of those committments and add those consequences as new constraints. Efficiency concerns drive us to look closely at the representation of constraints, inference procedures for solving constraints and deriving consequences, and the capture of inferred consequences as new constraints.

We report here on our current efforts at developing automated methods for deriving problem-specific constraint propagation code. This effort is part of a broader development of automated tools for transforming formal specifications into efficient and correct programs. The KIDS system [9] serves as the testbed for our experiments and provides tools for performing deductive inference, algorithm design, expression simplification, finite differencing, partial evaluation, data type refinement, and other transformations. We have used KIDS to derive over 60 algorithms for a wide variety of application domains, including scheduling, combinatorial design, sorting and searching, computational geometry, pattern matching, and mathematical programming.

A transportation scheduling application motivated our constraint propagation work [8]. We used KIDS semiautomatically to derive a global search (backtrack) scheduler. The derivation included inferring pruning conditions and deriving constraint propagation code. The resulting code is given in [8] and has proved to be dramatically faster than other programs running the same data (e.g. OPIS at CMU). The pruning and constraint propagation are so strong that the program does not backtrack on the data we have tried. For example, on a transportation problem involving 15,460 movement requirements obtained from the US Transportation Command, the scheduler produces a complete feasible schedule in about one minute. A constraint network formulation of this problem would have over 46,000 variables and 150,000 constraints.

Any constraint programming system depends on an abstract datatype of constraints. What are some of the basic operations on constraints? First, constraints are expressed in the language of some theory (e.g. linear arithmetic with inequalities) and there must be a representation of constraints[1]. Second, in that theory there must be procedures for solving constraints and extracting solutions. Third, there must be a procedure for inferring consequences of constraints and capturing their content in a new constraint (perhaps by weakening or approximation). Fourth, there must be a way to compose two constraints (e.g. to assimilate a new constraint).

Current constraint programming languages (e.g. CLP(R) [4], CHIP [2]) effectively carry out these operations via representations and operations specialized to the theory of the constraint language. For the sake of efficiency it may be necessary to design representations and operations that exploit not only the general background theory, but the intrinsic structure of the particular problem being solved.

*This research was supported in part by the Office of Naval Research under Contract Office of Naval Research Grant N00014-90-J-1733, in part by the Air Force Office of Scientific Research under Contract F49620-91-C-0073, and in part by DARPA/Rome Laboratories under Contract F30602-91-C-0043.

[1] We will not distinguish constraints and constraint sets, since a constraint set usually denotes the constraint that is a conjunction of the constituent constraints.

2 Global Search Theory

Our studies of constraint propagation take place within a formal model of a class of constraint programs called *global search* algorithms. Global search generalizes the well-known algorithm concepts of binary search, backtrack, and branch-and-bound [6].

The basic idea of global search is to represent and manipulate sets of candidate solutions. The principal operations are to *extract* candidate solutions from a set and to *split* a set into subsets. Derived operations include (1) *filters* which are used to eliminate sets containing no feasible or optimal solutions, and (2) constraint propagators that are used to eliminate nonfeasible elements of a set of candidate solutions. Global search algorithms work as follows: starting from an initial set that contains all solutions to the given problem instance, the algorithm repeatedly extracts solutions, splits sets, eliminates sets via filters and propagates constraints until no sets remain to be split. The process is often described as a tree (or DAG) search in which a node represents a set of candidates and an arc represents the split relationship between set and subset. The filters and constraint propagators serve to prune off branches of the tree that cannot lead to solutions.

The sets of candidate solutions are often infinite and even when finite they are rarely represented extensionally. Thus global search algorithms are based on an abstract data type of intensional representations called *space descriptors* (denoted by hatted symbols). The space descriptors can be thought of as constraints or representations of constraints.

Global search can be expressed axiomatically via a global search theory [6] which we elide here. In the following we present just those parts needed to discuss the formal derivation of constraint propagation code.

A problem can be specified by presenting an *input domain* D, an *output domain* R, and an *output condition* $O : D \times R \rightarrow boolean$. If $O(x, z)$ then we say z is a *feasible* solution with respect to input x. In other words, the output condition defines the conditions under which a candidate solution is feasible or acceptable. Global search theory extends the components of a problem with a datatype of *space descriptors* (constraint representations) $\hat{R}$ and a predicate $Satisfies : R \times \hat{R} \rightarrow boolean$ that gives the denotation of space descriptors; if $Satisfies(z, \hat{r})$ then z is in the set of candidate solutions denoted by $\hat{r}$ and we say that z satisfies $\hat{r}$.

One version of the transportation scheduling problem can be specified as follows. The input is a set of movement requirements and a collection of transportation resources. A movement requirement is a record listing the type of cargo, its quantity, port of embarkation, port of debarkation, due date etc. Schedules are represented as maps from resources to sequences of trips, where each trip includes earliest-start-time, latest-start-time, port of embarkation, port of debarkation, and manifest (set of movement requirements). The type of schedules has the invariant (or subtype characteristic) that for each trip, the earliest-start-time is no later than the latest-start-time. A partial schedule is a schedule over a subset of the given movement records.

Twelve constraints characterize a feasible schedule for this problem:

1. *Consistent POE and POD* – The POE and POD of each movement requirement on a given trip of a resource must be the same.
2. *Consistent Resource Class* – Each resource can handle only some movement types. For example, a C-141 can handle bulk and oversize movements, but not outsize movements.
3. *Consistent PAX and Cargo Capacity* – The capacity of each resource cannot be exceeded.
4. *Consistent Initial Time* – The start time of the first trip of a transportation asset must not precede its initial available date, taking into account any time needed to position the resource in the appropriate POE.
5. *Consistent Release Time* – The start time of a trip must not precede the available to load dates of any of the transported movement requirements.
6. *Consistent Arrival time* – The finish time of a trip must not precede the earliest arrival date of any of the transported movement requirements.
7. *Consistent Due time* – The finish time of a trip must not be later than the latest arrival date of any of the transported movement requirements.

8. *Consistent Trip Separation* – Movements scheduled on the same resource must start either simultaneously or with enough separation to allow for return trips. The inherently disjunctive and relative nature of this constraint makes it more difficult to satisfy than the others.

9. *Consistent Resource Use* – Only the given resources are used.

10. *Completeness* – All movement requirements must be scheduled.

These constraints are expressed concisely as quantified first-order sentences.

A simple global search theory of transportation scheduling has the following form. A set of schedules is represented by a partial schedule. The split operation extends the partial schedule by adding one movement requirement in all possible ways. The initial set of schedules is described by the empty partial schedule – a map from each available resource to the empty sequence of trips. A partial schedule is extended by first selecting a movement record *mvr* to schedule, then selecting a resource r, and then a trip t on r (either an existing trip or a newly created one). Finally the extended schedule has *mvr* added to the manifest of trip t on resource r. The alternative ways that a partial schedule can be extended naturally gives rise to the branching structure underlying global search algorithms. The formal version of this global search theory of scheduling can be found in [8].

When a partial schedule is extended it is possible that some problem constraints are violated in such a way that further extension to a complete feasible schedule is impossible. In global search algorithms it is crucial to detect such violations as early as possible. The next two subsections discuss two general mechanisms for early detection of infeasibility and techniques for mechanically deriving them.

2.1 Pruning Mechanisms

Pruning tests are derived in the following way. Let x be a problem input and $\hat{r}$ be a space descriptor. The test

$$\exists(z : R)\,(Satisfies(z, \hat{r}) \wedge O(x, z)) \tag{1}$$

decides whether there exist any feasible solutions satisfying $\hat{r}$. If we could decide this at each node of a global search algorithm then we would have perfect search – no deadend branches would ever be explored. In practice it would be impossible or horribly complex to compute it, so we rely instead on an inexpensive approximation to it. In fact, if we approximate (1) by weakening it (deriving a necessary condition of it) we obtain a sound pruning test. That is, suppose we can derive a test $\Phi(x, \hat{r})$ such that

$$\exists(z : R)\,(Satisfies(z, \hat{r}) \wedge O(x, z)) \Longrightarrow \Phi(x, \hat{r}). \tag{2}$$

By the contrapositive of (2), if $\neg\Phi(x, \hat{r})$ then there are no feasible solutions satisfying $\hat{r}$, so we can eliminate $\hat{r}$ from further consideration. More generally, necessary conditions on the existence of feasible (or optimal) solutions below a node in a branching structure underlie pruning in backtracking and the bounding and dominance tests of branch-and-bound algorithms [7].

It appears that the bottleneck analysis advocated in the constraint-directed search projects at CMU [3, 5] leads to a semantic approximation to (1) that is neither a necessary nor sufficient condition. Such a *heuristic* evaluation of a node is inherently fallible, but if the approximation to (1) is close enough it can provide good search control with relatively little backtracking.

In KIDS, a filter Φ is derived using a general-purpose first-order inference system. The inference of Φ takes place within the theory of the specified problem. Potentially, any special problem structure captured by the axioms and theorems of this theory can be exploited to obtain strong problem-specific pruning mechanisms. Analogous comments apply to the constraint propagation mechanisms discussed next. For details of deriving pruning mechanisms for various problems see [6, 9].

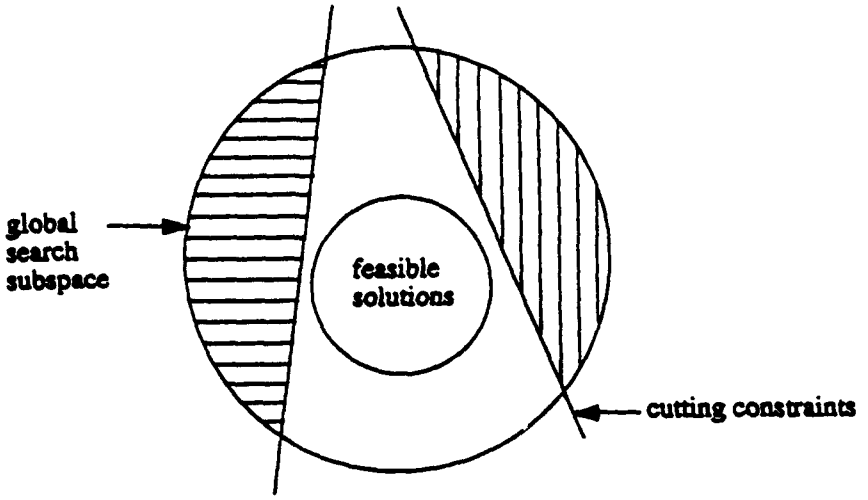


Figure 1: Global Search Subspace and Cutting Constraints

2.2 Cutting Constraints and Constraint Propagation

Pruning has the effect of removing a node (set of solutions) from further consideration. In contrast, constraint propagation has the effect of changing the space descriptor so that it denotes a smaller set of candidate solutions. Constraint propagation is based on the notion of *cutting constraints* which are necessary conditions $\Psi(x, z, \hat{r})$ that a candidate solution z satisfying $\hat{r}$ is feasible:

$$\forall(x : D, z : R, \hat{r} : \hat{R})\ (Satisfies(z, \hat{r}) \wedge O(x, z) \implies \Psi(x, z, \hat{r})). \tag{3}$$

By the contrapositive of (3), if $\neg\Psi(x, z, \hat{r})$ then z cannot be a feasible solution satisfying $\hat{r}$. So we can try to incorporate Ψ into $\hat{r}$ to obtain a new descriptor, without losing any feasible solutions. See Figure 1.

Once the inference system has been used to derive a cutting constraint Ψ, we specify an operation, called *Cut*, that maps $\hat{r}$ to a new descriptor $\hat{s}$ such that

$$Satisfies(z, \hat{s}) \iff (Satisfies(z, \hat{r}) \wedge \Psi(x, z, \hat{r})).$$

Constraint propagation is the iteration of Cut until we reach a fixpoint $Cut(\hat{t}) = \hat{t}$ (See Figure 2). The challenge in implementing constraint propagation is scheduling this iteration in order to minimize unnecessary work. This involves analyzing the dependencies between variables at design time and generating the control structure needed to reestablish a fixpoint when the *Split* operation causes the value of some variable to change.

The effect of constraint propagation is to propagate information through the subspace descriptor resulting in a tighter descriptor and possibly exposing infeasibility. There are several reasons for constraint propagation. First, it shrinks the space of candidate solutions and may thus reduce the branching required to explore it. Second, the generated descriptors may fail the pruning tests and thus allow early termination.

The mechanism for deriving cutting constraints is similar to (in fact a generalization of) that for deriving pruning mechanisms. For transportation scheduling, the derived Cut operation has the following form, where est_i denotes the earliest-start-time for trip i and est'_i denotes the earliest-start-time for trip i after applying *Cut* (analogously, lst_i denotes latest-start-time), and $roundtrip_i$ is the roundtrip time for trip i on resource r. For each resource r and the i^{th} trip on r,

$$est'_i = max \begin{cases} est_i \\ est_{i-1} + roundtrip_i \\ max\text{-}release\text{-}time(manifest_i) \end{cases}$$

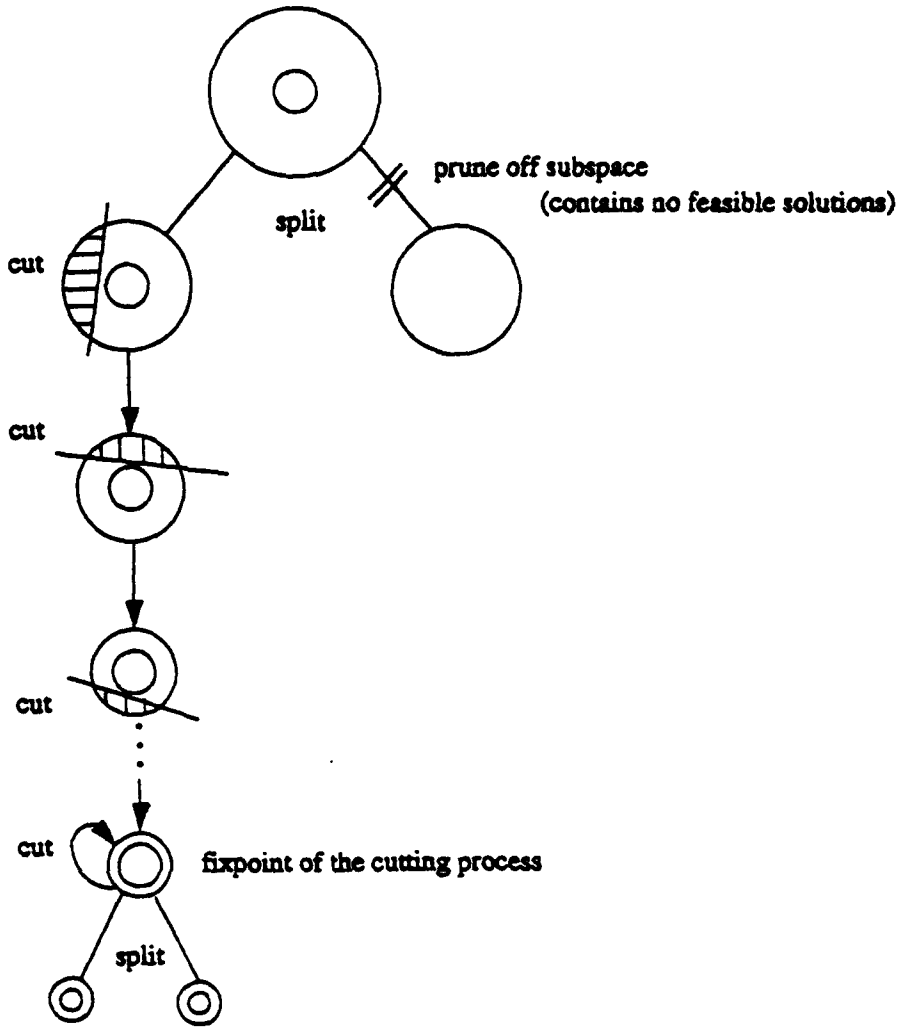


Figure 2: Pruning and Constraint Propagation

$$lst_i' = min \begin{cases} lst_i \\ lst_{i+1} - roundtrip_i \\ min\text{–}finish\text{–}time(manifest_i) \end{cases}$$

Here $max\text{–}release\text{–}time(manifest_i)$ computes the max over all of the release dates of movement requirements in the manifest of trip i and $min\text{–}finish\text{–}time(manifest_i)$ computes the minimum of the finish times of movement requirements in the same manifest. Boundary cases must be handled appropriately.

The effect of iterating this *Cut* operation after adding a new movement record to some trip will be to shrink the $(earliest\text{–}start\text{–}time, latest\text{–}start\text{–}time)$ window of each trip on the same resource. If the window becomes negative for any trip, then the partial schedule is necessarily infeasible and it can be pruned.

3 Summary

The message of this work is that there are knowledge-based tools that can be used to synthesize highly specialized and efficient implementations of constraint programs. The idea is to exploit not only the structure of the theory within which the constraints are stated, but the local theory of the particular problem being solved. The local structure supports the inference of specialized pruning and constraint propagation code. The use of a datatype refinement system [1] also allows specialized representations of objects, constraints, and efficient implementation of their operations. The result can be much more efficient code than is possible using general-purpose representations, solvers, and inference procedures.

References

[1] BLAINE, L., AND GOLDBERG, A. DTRE - a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 165–204.

[2] DINCBAS, M., VANHENTENRYCK, P., SIMONIS, H., AND AGGOUIN, A. The constraint logic programming language CHIP. In *Proceedings of the Second International Conference on Fifth Generation Computer Systems* (Tokyo, November 1988), pp. 249–264.

[3] FOX, M. S., SADEH, N., AND BAYKAN, C. Constrained heuristic search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (Detroit, MI, August 20–25, 1989), pp. 309–315.

[4] JAFFAR, J., MICHAYLOU, S., STUCKEY, P., AND YAP, R. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems 14*, 3 (July 1992), 339–395.

[5] SADEH, N. Look-ahead techniques for micro-opportunistic job shop scheduling. Tech. Rep. CMU-CS-91-102, Carenegie-Mellon University, March 1991.

[6] SMITH, D. R. Structure and design of global search algorithms. Tech. Rep. KES.U.87.12, Kestrel Institute, November 1987. to appear in *Acta Informatica*.

[7] SMITH, D. R. Structure and design of global search algorithms. Tech. Rep. KES.U.87.12, Kestrel Institute, November 1987.

[8] SMITH, D. R. Transformational approach to scheduling. Tech. Rep. KES.U.92.2, Kestrel Institute, November 1992.

[9] SMITH, D. R. KIDS - a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.

CONSTRAINT-BASED LANGUAGES FOR SCIENTIFIC DATABASE AND MODELING SYSTEMS.

Terence R. Smith and Keith Park.

22nd January, 1993.

1 GOAL

We focus our attention on the manner in which constraint programming (CP) may serve as a basis for languages that support the modeling and database activities of a large class of scientific investigators. In particular, we will examine the *applicability* of high-level languages based on constraints to problems relating to scientific modeling and database systems for large-scale environmental modeling. We believe that it is necessary to have a clear model of the nature of scientific investigations in order to proceed with such an examination. We therefore base our discussion of the applicability of such languages on a model of scientific investigation, and examine several classes of problems that arise as a consequence of the model.

Our discussion is based on the preliminary results of a multi-year investigation whose goal is the design and partial implementation of a modeling and database language with which scientists may represent a large variety of problems in such a manner that all irrelevant computational issues are hidden. In particular, our research is driven by the long-term computational requirements of a group of earth scientists who are investigating the hydrology, geochemistry and geomorphology of the entire Amazon drainage basin. The computational requirements of these scientists are based on the need to develop, in an iterative manner, models of complex phenomena. Such model development involves issues relating to very large, heterogeneous databases and to numerically intensive computations.

2 SCIENTIFIC INVESTIGATIONS, DOMAINS OF ELEMENTS AND CONSTRAINTS

The goal of our project is to provide scientists with a system that combines the functionality of a database system and the functionality of a modeling system. We believe, as a result of our requirements analysis of the Amazon project, that these two functionalities cannot be usefully separated. In particular, we believe that it is important to provide the scientist with a system that supports the iterative development, testing and application of *models* of phenomena under investigation. Such support is to be provided in terms of a high-level language that is being designed to enable scientists to represent, both efficiently and effectively, any of the operations that they may wish to carry out during the various stages of the conceptual or computational modeling of some phenomenon.

A central component in the design of the language is an extensible and potentially *very* large collection of abstract and concrete representational domains of elements that represent the total set of concepts that a scientist needs for modeling complex earth science phenomena in data- and numerically-intensive investigations. The reason for our focus on this concept of a set of representational domains is based upon our idea that the core activities of scientists involved in complex modeling operations relate to the discovery and application of large numbers of clever and powerful representational domains of elements and associated transformations. In particular, we view much of science as an activity in which scientific investigators:

1. construct, evaluate and employ a large set of *representational domains* for conceptual entities;
2. construct, evaluate and employ many representations of transformations between these domains;
3. construct instances of domain elements;
4. apply sequences of specific transformations to specific sets of domain elements in specific investigations.

Of particular importance are domains involving spatio-temporal elements.

There appear to be two major sets of reasons for supporting large lattices of domains and associated transformations as the core element in a high-level language that is designed to support scientific activity:

1. such a lattice reflects a major componenent of scientific activity, which involves the iterative discovery of *expressive* conceptual domains of elements and associated transformations, together with appropriate representations of these domains and transformations. The expressiveness of such domains is apparent in terms of their value in constructing models of given domains of phenomena. In many subfields of science and mathematics, there are typically large numbers of such domains. We therefore believe that it is important to provide computational support for the construction and use of such lattices of domains.
2. In computational terms, such a lattice is of value in relation to reducing the cost of search, insofar as domain membership may be used to provide information on where to search, and in terms of reducing errors, insofar as domain membership may be used to provide a basis for such operations as type checking.

2.1 Lattices of Domains and Constraints

Abstract representational domains correspond to the *conceptual* domains of a scientist while *concrete* representational domains correspond to the scientist's choice of a representation for the elements of some abstract domain, in terms of elements from other abstract and concrete representational domains. There may be more than one concrete representational domain for each abstract representational domain. The several characteristics of any concrete representational domain include:

1. the structure of the domain elements;
2. constraints on the values of domain elements;

3. sets of transformations on the domain elements, that in part provide the semantics of the domains.

Hence, one may view domain membership as involving constraints on the structure (or representation) of the elements in a domain; constraints on the values of the elements in a domain; and constraints on the transformations that are applicable to the elements in a domain. Furthermore, we note that a major mechanism for creating new representational domains involves placing constraints on the values of elements of previously constructed domains. The placing of such constraints also induces a structure on the domains that has inheritance properties associated with representation and structure.

3 CONSTRAINTS IN MODELING AND DATABASE LANGUAGES

Our basic hypothesis, based upon observations of the activities of many environmental scientists, is that nearly all of the problems for which such scientists require computational support may be expressed by constraints in a manner that is both very easy and very natural for the scientists. The ease of expression results, in part, from the facts that:

1. constraints are largely declarative in nature and typically permit a very parsimonious representation of conditions that some computational modeling activity must satisfy;

2. the iterative strengthening and weakening of constraints provides a simple yet powerful strategy for acquiring scientific understanding;

3. expressing problems in terms of constraints permits one to hide computational issues that are independent of the scientific phenomena under investigation.

More importantly, perhaps, the simplicity and expressiveness of constraint-based representations follows naturally from our model of scientific activity and in particular from its characterization in terms of lattices of algebraic domains. In particular, it follows from this model that there are at least four major classes of problems for which expression by means of constraints is natural and important;

1. checking for the satisfaction of constraints defining membership in some domain;

2. accessing elements of domains in terms of constraints;

3. accessing transformations in terms of constraints;

4. specifying transformations of elements in terms of constraints that relate to the schema of domain transformations and to values.

3.1 Checking Constraints that Define Membership in Some Domain

It is clear that one would frequently wish to check whether a given element belongs to some domain. For example, this would occur whenever type checking is being performed. In general, this will require that we check whether the value constraints associated with the domain are satisfied.

Another issue that arises with respect to value constraints that are used to define domains is that an arbitrary set of constraints may be unsatisfiable, unless there is some mechanism that tests for such unsatisfiability.

3.2 Accessing Elements of Domains

A major activity in many scientific investigations relates to accessing elements from given domains. One may view such access as a process in which constraints are placed on some abstract entity, which are satisfied during the access process. Given the previous comments on domains, it is clear that specifying that an element belongs to some domain is equivalent to specifying constraints concerning the representation, values and applicable transformations (i.e., the semantics) of the desired element. Additional constraints on the values of the element may then specify search within a given domain. An advantage of constraints is that is permits a *satisficing* strategy with respect to accessing appropriate domain elements, while it is also easy to refine a search with the addition of more constraints. There appears to be considerable value, from a scientist's point of view in terms of expressiveness, in representing values of domain elements in *implicit* form. For example, we may wish to represent constraints on the values of some elements in terms of equations or inequalities that must be solved in order to obtain explicit representations of the values. Computational systems must therefore possess some means for determining the meaning associated with such constraints.

3.3 Accessing Transformations of Domains

An issue that appears to have received little attention involves accessing *transformations on domains*. Apart from giving a names to transformations and accessing them by name, they too may be accessed by providing a set of constraints. Such constraints may be specified in terms of the schema of the constraint, expressed in terms of the domain names for the domain and range of the transformations, and by constraints on allowable subsets of domain-range pairs, which may be expressed in terms of arbitrary sets of constraints.

Accessing transformations using their names is very restrictive since the only feasible constraints on transformation names must be given in terms of syntactic processing of regular expressions in the name space. Moreover, even if transformation modules were given some mnemonic names suggestive of what the transformations do, we cannot expect globally acceptable naming conventions among the developers and users of transformations. Furthermore, since the values of input/output pairs of transformations are stored in its intensional form, value-based retrieval, while theoretically feasible, is impractical.

We therefore focus on signature-based retrieval of transformations. The domain names constitute partial information concerning transformations, and as such the signature of transformations may serve as search keys. For example, suppose that a domain called POLYGON is defined, and a transformation which computes intersections of two polygons is stored under the name *polygon_intersection*. If the signature of this transformation is stored in a relational table with the schema of [NAME, INPUT_DOM, OUTPUT_DOM], the names of the transformations may be retrieved by simple table lookup.

The use of domain names in the signature, however, will suffer from the same problem as the use of transformation names as search keys, since constraints on names are essentially that of syntactic identity. By relaxing the constraints in such a manner that the intended semantics of queries is preserved, transformations that are behaviorally identical could also be retrieved. Thus,

we could improve the situation by considering equivalence among domains and signatures. Aliasing or renaming of domains should be taken into consideration in processing queries, together with the order and any "irrelevant" domains appearing in the signature.

In addition to these syntactic relaxations, the system may perform some form of semantic matching of transformation signatures based on the notion of domain interdependencies. The class of interdependencies we will consider include equivalence, super/sub domains, aggregated domains, and other set-theoretic relations such as disjoint, covering, etc. To utilize these dependencies among domains, query processing may be extended by constraint solving capabilities. Suppose for example that a transformation-base is queried about transformations for computing the intersection of two rectangles. If there is no transformation whose signature matches exactly that of the query, the system relaxes the constraints and starts searching the domain lattice for its super domains. Since the domain RECTANGLE is a subdomain of POLYGON, together with constraints on the number of sides and angles between adjacent sides, transformation for computing the intersection of polygons may be accepted as an appropriate answer to the query.

3.4 Specifying Transformations of Elements in Terms of Constraints

An additional approach for accessing transformations involves the idea of implicitly defined transformations. When a transformation is not explicitly stored in the transformation-base, it may be of value for the system to suggest a construction of the desired transformation from the existing lattice of domains and the associated sets of transformations. Since transformation signatures may be represented in terms of a graph-like structure in which nodes denote domains and directed edges denote transformations, a path suggests the derivability of a new transformation and the edges in the path represent compositions of transformations.

As an illustration of the value of the proposed mechanism, consider a scientist investigating a model of river discharge who wishes to extract a representation of a drainage network (TN) from a representation of topography. Suppose that the only topographic dataset available for the region of interest takes the form of a triangulated irregular network (TIN). The scientist may query the system in order to locate appropriate transformations that transform the TIN to a DN with the use of a query that includes constraints on its signature. If there is no transformation in the system having this signature. there may be a transformation that generates a DN from a raster digital elevation model (DEM), and a further transformation that converts the TIN to a DEM, which when composed may provide an acceptable solution to the scientist's problem.

Set-based Concurrent Engineering

Allen C. Ward
Assistant Professor
Department of Mechanical Engineering and Applied Mechanics
University of Michigan
Ann Arbor, MI 48109-2125
alward@um.cc.umich.edu

Abstract: Concurrent engineering (CE) shortens design cycles and improves design quality by designing more of the total system, including both the product and the manufacturing system, in parallel. Reason and empirical studies suggest that members of the design team should communicate and reason about *sets* of possible solutions, rather than changes to a single solution. This process can in part be formalized and automated. This paper summarizes my recent work in this area.

1) Introduction

By design I mean the entire process by which an organization decides what to make and how to make it. My students and I focus on design processes that a) involve multiple designers using multiple models to make interconnected decisions simultaneously and b) gradually narrow a set of possible solutions, rather than making changes to a "points in the design space." A trivial example is arranging a meeting by telephone. We may call with a single time, changing it as we discover conflicts. Or, we may identify several possible days, and call to eliminate unsatisfactory times; this set-based approach is superior when the problem is highly constrained.

This discussion proceeds from motivations, through some basic concepts, to recent results, and finally to current projects.

1) Motivation

If team members ensure that their communications are correct about the entire set of solutions to be considered, then communications remain valid throughout the design process. This permits simultaneous decisions. Conversely, communication that change a single solution invalidate previous communications and decisions. Nevertheless, engineers often emphasize "shortening the iterative loop", and design theorists often emphasize rapid translation of point solutions.

These point-based views may be influenced by academic instruction and optimization theory, which emphasizes iterative hill-climbing using a single model: experience with CE should lead to a more set-based approach. Jeff Liker, a sociologist in the UM Industrial Operations Engineering department, and I are finding support for this hypothesis by comparing US and Japanese automobile companies. For example, we have studied the design of cooling fans by suppliers. All assembly companies studied but one communicate with suppliers through early, informal discussions, followed by a "hard specification". Suppliers design a part to meet the specification. The exception, a highly successful Japanese company with 30 years experience in CE, imposes a 20-30 percent "design tolerance" on specifications. The suppliers, in turn, produce some 30 different prototype designs. The specifications and solutions are gradually narrowed: a five percent "design tolerance" remains two years into the development cycle. This is clearly expensive, but increases reliability of communication, design performance and modularity, flexibility in response to concurrent market research, and availability of data for the next design cycle.

We have begun incorporating these ideas, together with a theory of design relationships that provides a fundamental explanation for most of the standard CE "methodologies", into teaching design at the university and in industry. Performance in the UM senior mechanical design project

course has been improving dramatically, although that causes cannot be isolated. The Industrial Technology Institute has agreed to assume the training load.

Additional motivation is provided by the success of a "mechanical design compiler," a program that accepts schematics, specifications, and utility functions for a fairly wide variety of mechanical designs, and returns an optimal selection of catalog numbers for components implementing the design. The compiler, part of my PhD work at the MIT AI Lab, appears to have been the first of its kind.

We also apply these concepts in designing innovative, computer-controlled machines. While most computer-controlled machines evolved from manual machines, or imitation of humans, we examine the design space to find new approaches that exploit computer control to simplify the mechanical system. We also hope that the set-based approach, and the associated theory of design relationships, will support construction of "maps of the design space" that can be communicated in journal articles, removing an important obstacle to treating design as academic research. Currently funded projects include a milling machine and a transfer press (consulti project); proposals are in preparation for an internal combustion engine, an autonomous vel and two actuators: one with high stiffness to inertia ratio, one with high power to weight rati

2) Basic concepts

The overall goal of this research is a computationally implemented theory, comparable to optimization theory, but supporting multiple designers *(agents)* using multiple models and making decisions simultaneously. Where game theory focuses on the conflict of objectives among agents, this theory focuses on the conflict of information, prescribing local decisions based on partial information.

Early research created a formalism, called the Labeled Interval Calculus (LIC), for concurrent reasoning over networks of ternary relationships about the feasibility of sets of possibilities. The calculus extends the basic notion of interval constraint propagation (or interval arithmetic) in several ways: 1) It adds "inverse" operations. 2) It uses "labels" to assign various meanings to intervals. 3) It combines these into inference rules that produce provably correct conclusions about feasibility.

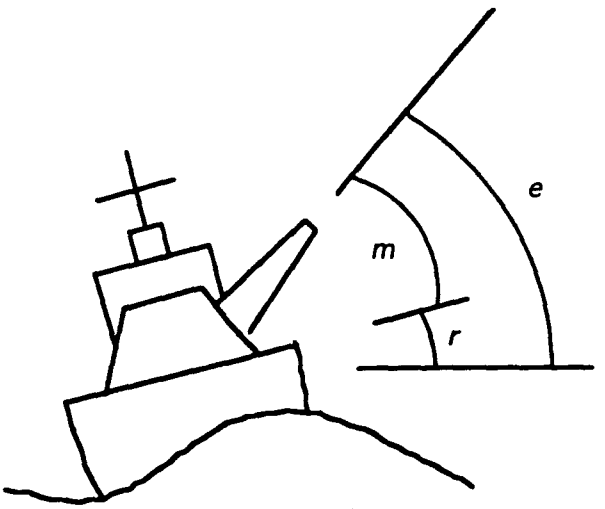


Figure: Designing a gun mount

These ideas can be understood through an example. Suppose we are designing mounts for an old-fashioned naval gun (figure). We want the gun to aim steadily at *every* elevation e (relative to the horizon) between 0 and 40 degrees, even though the ship may be rolling at any angle r (also relative the horizon) between -20 and 20 degrees. Thus, the mount angle m must move from -20 through 60 degrees.

To formalize this inference, let E be the interval of values for e, written $<e\ 0\ 40>$, $R = <r\ -20\ 20>$ and let G be the relationship $e + r - m = 0$. Define Range(G,E,R) to produce the interval of values for m which is compatible with E and R under the relationship G; that is M =

$\{m \mid \exists e \in E, \exists r \in R$ satisfying $G\} = \langle m$ -20 60$\rangle$. This is (for this "monotonic" equation) just interval propagation or interval arithmetic in the usual sense. We can perform the operation by substituting values for the endpoints of E and R into G, then selecting the minimum and maximum of the results as the endpoints of M.

e	+	r	=	m				
0	+	-20		-20				
0	+	20			20			
40	+	-20				20		
40	+	20					60	
Range(G,E,R) =		M =		$\langle m$	-20		60$\rangle$	

The operation defined, we can formulate the inference rule:

every $\langle E \rangle$ & *every* $\langle R \rangle$ & $G \Rightarrow$ *every* Range(G,E,R), where the rule is general, but stated for ease of interpretation in symbols specific to this problem. In the mechanical compiler, equations were linked into a full network based on the schematic, and each inference often triggered more.

Now to invert the problem. Suppose we have a mount design capable of producing angles m from -20 to 60, and want to know what elevations we can fire despite a -20 to 20 degree roll. We need another operation, Domain(G,M,R) = $\{e \mid \forall r \in R$, $\exists m \in M$ satisfying $G\}$. Domain selects the two central results of the endpoint substitution.

m	-	r	=	e					
-20	-	20		-40					
-20	-	-20			0				
60	-	20				40			
60	-	-20							80
Domain(G,M,R) =		E =			$\langle e$	0	40	$\rangle$	

In fact, Domain(G,X,Y) = Z if and only if Range(G,Y,Z) = X. Domain is an inverse to Range, in much the sense that division is a inverse to multiplication. Of course, we also need another inference rule:

every $\langle R \rangle$ & *only* $\langle M \rangle$ & $G \Rightarrow$ *only* Domain(G,R,M).

Neither inference is always correct: the second, for example, depends on m and e varying independently, "causing" the variation in r. The mechanical design compiler used another level of labels indicating whether variables were fixed at manufacture or changed during operations, as well as some useful but ad hoc assumptions about the problems, to address this issue; current research seeks a formal representation for such causal reasoning. At least three kinds of "causality" turn out to be relevant.

There is another useful "inverse" we call Sufficient-Points, defined by Sufficient-Points(G, M, E) = $\{r \mid$ Range(G,$\{r\}, E) \supseteq M\}$. A total of 16 propagation rules, together with abstraction and elimination rules, were used in the compiler.

3. Recent results

Nothing in the formulation of the operations or inferences above restricts them to the algebraic, monotonic equations covered by the LIC. Recent work generalizes them.

Bain's master's thesis (1992) extends the operations to all convex ternary equations, as required for some mechanical systems, demonstrating that monotonicity is not required.

Chen's PhD thesis (1992) extends the operations from algebraic to matrix equations. For matrix/vector equations of the form **Ax=b**, the three fundamental operations have a total of twelve variants. The interval matrix arithmetic community, an applied mathematics community with some 2000 published papers, had independently discovered 5 of the twelve variants, but apparently has not identified the general forms. This suggests that the LIC may represent a new "twig" of mathematical research.

Chen applied these processes to simple finite element problems, suggesting that it may be possible to use finite element methods not to analyze a completed design, but to reason about the feasible designs. He is now applying his work to robust control system design.

Lin's PhD thesis (1992) inductively extends the LIC operations to work on sets of intervals (represented by "quadrevals"). Propagation operations are defined on quadrevals, whose elements are connected by interval relationships defined using the LIC operations. A second extension is to "octervals": sets of sets of intervals. These higher order constructs turn out to be equivalent, without labels, to the labeled intervals, and each operation on them to several LIC inferences. They therefore provide an equivalent formulation for the LIC, which can be used to establish completeness of the LIC. These results then can be used to demonstrate that the LIC cannot solve the "causality problem" mentioned above, without introducing new concepts.

4. Current and near term research

While my ultimate goal is an implemented axiomatization, I lack John von Neumann's skills and proceed through mutually supporting sub-projects (concurrently, of course.)

a) The national Automated Configuration Design System (ACDS). Working programs provide insight. This large scale implementation of the distributed set-based design concept is led by Bill Birmingham (of the UM EECS department), and includes Chelsea White (Chairman, Industrial and Operations Engineering), Dan Atkins (Dean, Information and Library Science), and Mike Wellman and Edward Durfee of EECS. Like the MDC, ACDS will accept a schematic and other specifications, and select implementing components (initially from catalogs, later including parametrically designed components). However, components may be mechanical, electronic, and software; nationwide vendors will "bid" over electronic networks, supporting "agile manufacturing". ACDS will test the following extensions to theory.

b) Causal reasoning in design: discussed above.

c) Extension of interval based methods to reasoning about utility. The LIC can only eliminate provably infeasible solutions; the MDC used centralized optimization methods. Here, each interface variable between problem components will be controlled by one of the agents involved. Other agents will provide intervals of possible marginal utility on the variable; intervals are required because the marginal utility depends in part on decisions not yet made. Chelsea White's Imprecisely Specified Multi-Attribute Utility Theory will be modified to allow the deciding agent to eliminate Pareto-dominated solutions. LIC extensions will prove that at least one of the dominant solutions will be feasible.

d) Stochastic decision procedures. The previous mechanisms cannot guarantee convergence; the stochastic approaches considered here can. Controlling agents will transmit probability distributions on their selection of interface values. They will use genetic algorithms (and other standard methods) to achieve "conceptual robustness" against the "conceptual noise" imposed by their uncertainty about the decisions of other team members. They will consider the value of time in determining whether to eliminate designs with utilities strongly dependent on other's decisions.

e) Axiomatization. A more fundamental, broader approach will assume a local perspective, and explicitly represent the sources of variation, to provide a fully axiomatic discussion of the actions appropriate for an agent operating in a concurrent engineering network.

Constraint Programming in Constraint Nets

Ying Zhang
Department of Computer Science
University of British Columbia
Vancouver, B.C.
Canada V6T 1Z2
zhang@cs.ubc.ca

Alan K. Mackworth *
Department of Computer Science
University of British Columbia
Vancouver, B.C.
Canada V6T 1Z2
mack@cs.ubc.ca

Abstract

We view constraints as relations and constraint satisfaction as a dynamic process of approaching a stable equilibrium. We have developed an algebraic model of dynamics, called Constraint Nets, to provide a real-time programming semantics and to model and analyze dynamic systems. In this paper, we explore the relationship between constraint satisfaction and constraint nets by showing how to implement various constraint methods on constraint nets.

1 Motivation

Constraints are relations among entities. Constraint satisfaction can be viewed in two different ways. First, in the logical deductive view, a constraint system is a structure $\langle D, \vdash \rangle$, where D is a set of constraints and $\vdash$ is an entailment relation between constraints [20]. In this view, constraint satisfaction is seen as a process involving multiple agents concurrently interacting on the store-as-constraint system by checking entailment and consistency relations and refining the system monotonically. This approach is useful in database or knowledge-based systems, and can be embedded in logic programming languages [2, 5, 9]. Characteristically, the global constraint is not explicitly represented, even though for any given relation tuple the system is able to check whether or not it is entailed.

An alternative view, more appropriate for real-time embedded systems, is to formulate the constraint satisfaction problem as finding a relation tuple that is entailed by a given set of constraints [12]. In this paper, we present an approach to this problem. In this approach, constraint satisfaction is a dynamic process with each solution as a stable equilibrium, and the solution set as an attractor of the process. "Monotonicity" is characterized by a Liapunov function, representing the "distance" to the set of solutions over time. Moreover, soft as well as hard constraints can be represented and solved. This approach has been taken in neural nets [18], optimization, graphical simulation [16] and robot control [15]; however, it has not yet been investigated seriously in the area of constraint programming.

We have developed and implemented an algebraic model of dynamics, called Constraint Nets (CN), to provide a real-time programming semantics [22] and to model and analyze robotic systems [23]. Here we investigate the relationship between constraint satisfaction and constraint nets. The rest of this paper is organized as follows. Section 2 describes some basic concepts of dynamic systems. Section 3 introduces Constraint Nets. Section 4 presents various constraint methods for solving global consistency and unconstrained optimization problems. Section 5 discusses embedded constraint solvers and some implementation issues. Section 6 concludes the paper.

2 Properties of Dynamic Systems

In this section, we review some basic concepts in metric spaces, dynamic systems and the relationship among stability, attractors and Liapunov functions.

*Shell Canada Fellow, Canadian Institute for Advanced Research

2.1 Metric spaces

Let $\mathcal{R}$ be the set of all real numbers and $\mathcal{R}^+$ denote the set of all nonnegative real numbers. A *metric* on a set X is a function $d : X \times X \rightarrow \mathcal{R}^+ \cup \{\infty\}$ which satisfies the following axioms for all $x, y, z \in X$:

1. $d(x, y) = d(y, x)$.
2. $d(x, y) + d(y, z) \geq d(x, z)$.
3. $d(x, y) = 0$ iff $x = y$.

A *metric space* is a pair $\langle X, d \rangle$ where X is a set and d is a metric on X. In a metric space, $d(x, y)$ is called "the distance between x and y". Given a metric space, we can define the distance between a point and a set of points as: $d(x, X^*) = \inf_{x^* \in X^*} d(x, x^*)$.

For any point $x^* \in X$ and $\epsilon > 0$, if $\{x | d(x, x^*) \leq \epsilon\} \supset \{x^*\}$, we call this set the ϵ-neighborhood of x^*, denoted $N^\epsilon(x^*)$. Similarly, for any subset $X^* \subset X$, if $\{x | d(x, X^*) \leq \epsilon\} \supset X^*$, we call this set the ϵ-neighborhood of X^*, denoted $N^\epsilon(X^*)$.

Let $\mathcal{T}$ be a set of totally ordered time points which can be either discrete or continuous. A *trace* $v : \mathcal{T} \rightarrow X$ is a function from a set of time points to a set of values. We use $\mathcal{V}_\mathcal{T}^X$ to denote the set of all traces from $\mathcal{T}$ to X. Given a metric space $\langle X, d \rangle$ and a trace v, a trace v *approaches a value* $x^* \in X$ iff $\lim_{t \rightarrow \infty} d(v(t), x^*) = 0$; v *approaches a set* $X^* \subset X$ iff $\lim_{t \rightarrow \infty} d(v(t), X^*) = 0$.

2.2 Dynamic systems

The term *dynamic* refers to phenomena that produce time-changing patterns, the characteristics of the pattern at one time being interrelated with those at other times [10]. A *process* $p : X \rightarrow \mathcal{V}_\mathcal{T}^X$ is a function from a set of values X to a set of traces $\mathcal{V}_\mathcal{T}^X$. Intuitively, p characterizes a set of traces which are solely determined by their initial values. We use $\phi_p(x)$ to denote the set of values in the trace of $p(x)$, i.e. $\phi_p(x) = \{p(x)(t) | t \in \mathcal{T}\}$.

A point $x^* \in X$ is an *equilibrium (or fixpoint) of the process* p iff $\forall t, p(x^*)(t) = x^*$. An equilibrium x^* is *stable* [13] iff $\forall N^\epsilon(x^*) \exists N^\delta(x^*) \forall x \in N^\delta(x^*) \phi_p(x) \subseteq N^\epsilon(x^*)$.

A set $X^* \subset X$ is an *attractor* [19] of the process p iff $\exists N^\epsilon(X^*) \forall x \in N^\epsilon(X^*) \lim_{t \rightarrow \infty} d(p(x)(t), X^*) = 0$; X^* is an *attractor in the large* iff $\forall x \in X \lim_{t \rightarrow \infty} d(p(x)(t), X^*) = 0$. If $\{x^*\}$ is an attractor (in the large) and x^* is a stable equilibrium, x^* is called an *asymptotically* stable equilibrium (in the large).

2.3 Liapunov functions

Let $X^* \subset X$ and $\Omega = N^\epsilon(X^*)$ for some $\epsilon > 0$. A *Liapunov function* for X^* and a process $p : X \rightarrow \mathcal{V}_\mathcal{T}^X$ is a function $V : \Omega \rightarrow \mathcal{R}$, satisfying:

1. $\forall x, x' \in \Omega, V(x) \leq V(x')$ iff $d(x, X^*) \leq d(x', X^*)$,
2. $\forall x \in \Omega \forall t, V(p(x)(t)) \leq V(x)$.

The first condition states that V has a local minimum at $x^* \in X^*$. The second condition guarantees that V moves downhill along any traces of p starting at $x \in \Omega$. This definition is a simplified version of the one given in [10]. The following two theorems are similar to those in [10].

Theorem 1 *An equilibrium $x^* \in X$ of a process p is stable if there exists a Liapunov function V for $\{x^*\}$ and p.*

Proof: Let Ω be the domain of V, which is an ϵ'-neighborhood of x^* for some $\epsilon' > 0$. Given an ϵ-neighborhood $N^\epsilon(x^*)$ of x^*, let $\delta = min(\epsilon, \epsilon')$, we have a δ-neighborhood $N^\delta(x^*) \subseteq \Omega$, therefore, $\forall x \in N^\delta(x^*) \phi_p(x) \subseteq N^\epsilon(x^*)$. □

Theorem 2 *Suppose a process p satisfies the following condition: for any $x^* \in X$, if there is x such that $p(x)$ approaches x^*, then x^* is an equilibrium. A set of stable equilibria $X^* \subset X$ of the process p is an attractor if there exists a Liapunov function V for X^* and p, such that V satisfies the following conditions:*

1. *V is continuous, i.e. $d(x, x') \to 0$ implies $|V(x) - V(x')| \to 0$.*

2. *$\forall x \in \Omega \forall t, V(p(x)(t)) < V(x)$ if $x \notin X^*$.*

Furthermore, if $\Omega = X$, X^* is an attractor in the large.

Proof: For any $x \in \Omega$, since V is continuous, $\lim_{t\to\infty} V(p(x)(t)) = V(\lim_{t\to\infty} p(x)(t)) = V(x^*)$. We can prove that $x^* \in X^*$ since otherwise according to Condition 2, x^* cannot be an equilibrium. If $\Omega = X$, X^* is an attractor in the large. □

3 Constraint Nets: A General Model of Dynamics

In this section, we first introduce Constraint Nets, a model for dynamic systems, then examine the relationship between constraint nets and constraint satisfaction.

3.1 Constraint Nets

A constraint net is composed of transductions and locations. A *transduction* is any mapping from a tuple of input traces to an output trace which is causal, viz. the output value at any time is determined by the input values prior to or at that time. Transductions are mathematical models of transformational processes.

There are two elementary types of transductions for dynamic systems: transliterations and delays. A *transliteration* $f_{\mathcal{T}}$ is a pointwise extension of a function f over a time set $\mathcal{T}$. We use $\delta(init)$ and $\int(init)$ to denote a *unit delay* in a discrete system and an *integration* in a continuous system respectively with *init* as the initial value.

A *constraint net* is a triple $CN \equiv (Lc, Td, Cn)$, where Lc is a set of *locations*, Td is a set of *transductions*, each of which is associated with a tuple of *input ports* and an *output port*, Cn is a set of directed *connections* between locations and ports of transductions, with the following restriction: (1) there is at most one connection pointing to each location, (2) each port of a transduction connects to a unique location and (3) no location is isolated.

A location is an *input* if there is no connection pointing to it otherwise it is an *output*. A constraint net is *closed* if it has no input location otherwise it is *open*. We use $CN(I, O)$ to denote a *module* with a set of input locations I and a set of output locations O as the interface.

The graphical representation of a constraint net is a bipartite directed graph where locations are represented by circles, transductions by boxes and connections by arcs, each from a port of a transduction to a location or vice versa.

Semantically, each location l denotes a trace x and each transduction $F_{\mathcal{T}}$ corresponds to an equation $x = F_{\mathcal{T}}(x_1, \ldots, x_n)$. A constraint net corresponds to a set of equations; its semantics is the least fixpoint of the equation set [22].

In general, constraint nets can model hybrid dynamic systems, with components operating on different time structures or triggered by events. In this paper, we focus on only two types of constraint nets: discrete transition systems and continuous integration systems, corresponding respectively to two different types of constraint solvers.

3.2 Constraint solvers

An output location is a *state* location if it is an output of a unit delay or an integration. A state of a constraint net is a mapping from the set of state locations to a set of values.

A *constraint solver* can be regarded as a special kind of constraint net which is closed and state-determined, i.e. a state trace is determined by its initial state. A constraint solver CS defines a process $p : S \to \mathcal{V}_{\mathcal{T}}^S$ where S is a set of states and $\mathcal{V}_{\mathcal{T}}^S$ is a set of state traces. A (stable) equilibrium of p is called a (stable) equilibrium of CS; an attractor of p is called an attractor of CS.

CS solves C iff (1) every solution of C is a stable equilibrium of CS and (2) the solution set of C is an attractor of CS; *CS solves C globally* iff, in addition, the solution set of C is an attractor in the large.

Lemma 1 *If CS solves C globally then every equilibrium of CS is a solution of C.*

Proof: Trivial. □

We discuss here two basic types of constraint solvers: state transition systems for discrete cases and state integration systems for continuous cases. A *state transition system* is a pair $\langle S, f \rangle$ where S is the set of states and $f : S \rightarrow S$ is the *state transition function*. A state transition system can be represented by a constraint net with a transliteration f_T and a unit delay $\delta(s_0)$ where $s_0 \in S$ is an initial state (Fig. 1). The solution of this net is an infinite sequence $p(s_0) \equiv s_0, f(s_0) \ldots f^n(s_0) \ldots$.

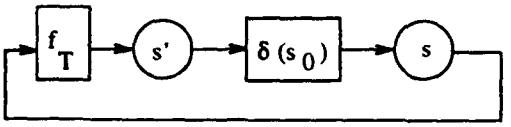


Figure 1: A constraint net representing $\langle S, f \rangle$

Clearly, a state $s^* \in S$ is an equilibrium of a state transition system $\langle S, f \rangle$ iff $s^* = f(s^*)$.

Lemma 2 *Let $\langle S, d \rangle$ be a metric space. An equilibrium s^* in a state transition system is stable if $\exists \Omega = N^\epsilon(s^*), \forall s \in \Omega, d(f(s), f(s^*)) \leq d(s, s^*)$. Moreover, s^* is asymptotically stable if $\exists \Omega, \forall s \in \Omega, d(f(s), f(s^*)) \leq kd(s, s^*)$ for $0 \leq k < 1$. If $\Omega = S$, s^* is an asymptotically stable equilibrium in the large.*

Proof: Let $V(s) = d(s, s^*)$. It is easy to see that V is a Liapunov function for s^* and $\langle S, f \rangle$. □

Integration is a basic transduction on continuous time structures. A *state integration system* is a differential equation $\frac{ds}{dt} = f(s)$[1] which can be represented by a constraint net with a transliteration f_T and an integration $\int(s_0)$ where $s_0 \in S$ is an initial state (Fig. 2).

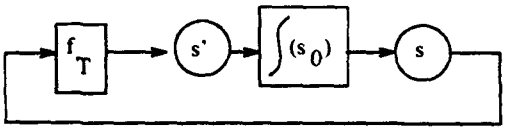


Figure 2: A constraint net representing $\frac{ds}{dt} = f(s)$

Clearly, a state $s^* \in S$ is an equilibrium of a state integration system $\frac{ds}{dt} = f(s)$ iff $f(s^*) = 0$.

Lemma 3 *An equilibrium s^* in a state integration system is stable if f is continuous at s^* and s^* is a local minimum of $-\int f(s)ds$. Moreover, s^* is asymptotically stable if it is the unique minimum in its neighborhood. If there is no other equilibrium and s^* is the global minimal point, s^* is an asymptotically stable equilibrium in the large.*

Proof: Let $V(s) = -\int f(s)ds$. It is easy to check that (1) $\frac{dV}{dt} \leq 0$ and (2) since s^* is a local minimum of V, there is a neighborhood of s^* such that $V(s) \leq V(s')$ iff $d(s, s^*) \leq d(s', s^*)$. Therefore V is a Liapunov function for s^* and $\frac{ds}{dt} = f(s)$. □

4 Properties of Constraint Methods

In this section, we examine various constraint methods and their properties. In particular, we discuss two types of constraint satisfaction problems, namely, global consistency and unconstrained optimization, for four classes of relations: relations on finite domains, linear, convex and nonlinear relations in n-dimensional Euclidean space $\langle \mathcal{R}^n, d_n \rangle$, where $d_n(x, y) = |x - y| = \sqrt{(x_1 - y_1)^2 + \ldots (x_n - y_n)^2}$.

Global consistency corresponds to solving hard constraints and *unconstrained optimization* corresponds to solving soft constraints. A problem of the first kind can be translated into one of the second by introducing an energy function representing the degree of global consistency.

[1] If s is a tuple, $\frac{ds}{dt} = f(s)$ represents a set of equations.

4.1 Unconstrained optimization

The problem of *unconstrained optimization* is to minimize an energy function $\mathcal{E} : \mathcal{R}^n \rightarrow \mathcal{R}$. Here we first discuss two methods for this problem: the gradient method (GM) [16] and Newton's method (NM) [19], and then study the schema model (SM) for minimizing an energy function $\mathcal{E} : [0,1]^n \rightarrow \mathcal{R}$.

4.1.1 Gradient method

The gradient method is based on the *gradient descent* algorithm, where state variables slide downhill in the opposite direction of the gradient. Formally, if the function to be minimized is $\mathcal{E}(x)$ where $x = \langle x_1, \ldots, x_n \rangle$, then at any point, the vector that points towards the direction of maximum increase of $\mathcal{E}$ is the gradient of $\mathcal{E}$. Therefore, the following gradient descent equations model the gradient method:

$$\frac{dx_i}{dt} = -k_i \frac{\partial \mathcal{E}}{\partial x_i}, \quad k_i > 0. \tag{1}$$

Let $X^* = \{x | x^* \text{ is a true local minimum of } \mathcal{E}\}$[2] and $Y^* = \{x^* | x^* \text{ is a global minimum of } \mathcal{E}\}$. Let GM be a constraint net representing the gradient descent equations (1). The following theorem specifies the conditions under which GM solves the problems X^* and Y^*.

Theorem 3 *GM solves X^* if $\frac{\partial \mathcal{E}}{\partial x}$ is continuous at every $x^* \in X^*$. GM solves Y^* if, in addition, $\mathcal{E}$ is bounded from below. GM solves Y^* globally if, in addition, $\mathcal{E}$ is convex*[3].

Proof: According to Lemma 3 every solution is a stable equilibrium. According to Theorem 2, by letting $\mathcal{E}$ be a Liapunov function, we can prove that X^* is an attractor. If $\mathcal{E}$ is convex, Y^* contains all of the equilibria, therefore, the set of global minima is the attractor in the large. □

4.1.2 Newton's method

Newton's method is to minimize a second-order approximation of the given energy function, at each iterative step. Let $\Delta\mathcal{E} = \frac{\partial \mathcal{E}}{\partial x}$ and J be the Jacobian of $\Delta\mathcal{E}$. At each step with current point $x^{(k)}$, Newton's method is to minimize the function:

$$\mathcal{E}_a(x) = \mathcal{E}(x^{(k)}) + \Delta\mathcal{E}^T(x^{(k)})(x - x^{(k)}) + \frac{1}{2}(x - x^{(k)})^T J(x^{(k)})(x - x^{(k)}).$$

Let $\frac{\partial \mathcal{E}_a}{\partial x} = 0$, we have:

$$\Delta\mathcal{E}(x^{(k)}) + J(x^{(k)})(x - x^{(k)}) = 0.$$

The solution of the above equation becomes the next point, i.e.

$$x^{(k+1)} = x^{(k)} - J^{-1}(x^{(k)})\Delta\mathcal{E}.$$

Newton's method defines a state transition system $\langle \mathcal{R}^n, f \rangle$ where $f(x) = x - J^{-1}(x)\Delta\mathcal{E}(x)$.

Let NM be the constraint net representing Newton's method. The following theorem specifies the conditions under which NM solves the problem X^* and Y^*.

Theorem 4 *NM solves X^* if $|J(x^*)| \neq 0$ at every $x^* \in X^*$, i.e. $\mathcal{E}$ is strictly convex at x^*. NM solves Y^* if, in addition, $\mathcal{E}$ is bounded from below. NM solves Y^* globally if, in addition, $\mathcal{E}$ is convex.*

Proof: First, we prove that $x^* = f(x^*)$ and $|J(x^*)| \neq 0$ implies x^* is asymptotically stable. Let R be the Jacobian of f. It is easy to check that $|R(x^*)| = 0$. There exists a neighborhood of x^*, $N^\epsilon(x^*)$, for any $x \in N^\epsilon(x^*)$, $|f(x) - f(x^*)| \leq k|x - x^*|$ for $0 < k < 1$. According to Lemma 2, x^* is asymptotically stable. If $\mathcal{E}$ is convex, there is no other equilibrium, so that x^* is asymptotically stable in the large. □

Here we assume that the Jacobian and its inverse are obtained off-line. Newton's method can also be used to solve a set of nonlinear equations $g(x) = 0$ by replacing $\Delta\mathcal{E}$ with g.

[2] This excludes flat maximum and saddle surfaces.

[3] A function f is convex iff for any $\lambda \in (0,1)$, $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$; it is strictly convex iff the inequality is strict. Obviously, a strictly convex function has a unique minimal point. Linear functions are convex, but not strictly convex. A quadratic function $x^T M x + c^T x$ is convex if M is semi-positive definite; it is strictly convex if M is positive definite.

4.1.3 Schema model

The schema model has been used for finite constraint satisfaction in the PDP framework [18]. Basically, there is a set of units, each can be on or off; constraints between units are represented by weights on connections. The energy is typically a quadratic function in the following form:

$$\mathcal{E}(a) = -(\Sigma_{i,j} w_{ij} a_i a_j + \Sigma_i b_i a_i) = -(a^T W a + b^T a)$$

where $a_i \in [0,1]$ indicates the activation value and b_i specifies the bias for unit i, w_{ij} represents the constraint between two units i and j. w_{ij} is positive if units i and j support each other, it is negative if the units are against each other and it is zero if the units have no effect on each other.

There are various methods for solving this problem. The schema model [18] provides the simpliest discrete relaxation method. Let $n_i(a) = \frac{\partial \mathcal{E}}{\partial a_i} = -\Sigma_j w_{ij} a_j - b_i$. The schema model defines a state transition system $\langle [0,1]^n, f \rangle$ where $f = \langle f_1, \ldots f_n \rangle$ with $f_i(a) = a_i - n_i(a) a_i$ if $n_i(a) > 0$ and $f_i(a) = a_i - n_i(a)(1 - a_i)$ otherwise. In other words, $f_i(a) = (1 - |n_i(a)|)a_i - min(0, n_i(a))$.

Theorem 5 *Let SM be a constraint net representing the schema model with $|n_i(a)| \leq 1$ for any i and a. SM solves the set of minima of $\mathcal{E}$, denoted A^*.*

Proof: Let $a^{(k+1)}$ denote $f(a^{(k)})$. First, because $|n_i(a)| \leq 1$, $a^{(k)} \in [0,1]^n$ implies $a^{(k+1)} \in [0,1]^n$. Therefore f is well defined. Second, for each minimum a^* of $\mathcal{E}$, and for any i, either (1) $n_i(a^*) = 0$ or (2) $n_i(a^*) > 0$ and $a_i^* = 0$ or (3) $n_i(a^*) < 0$ and $a_i^* = 1$. Therefore a^* is an equilibrium. Now we prove that a^* is stable. Let $N^\epsilon(a^*)$ be a neighborhood such that $\forall a \in N^\epsilon(a^*)$ and for any i if $n_i(a^*) \neq 0$, then $n_i(a)$ and $n_i(a^*)$ have the same sign otherwise if $n_i(a) \geq 0$, then $a_i \geq a_i^*$ otherwise $a_i \leq a_i^*$. Such a neighborhood exists because n_i is continuous. Considering $|f_i(a) - a_i^*|$, there are four cases.

1. $n_i(a^*) > 0$: In this case, $a_i^* = 0$ and $|f_i(a) - a_i^*| = |f_i(a)| = |1 - n_i(a)| \times |a_i| \leq |a_i - a_i^*|$.
2. $n_i(a^*) < 0$: In this case, $a_i^* = 1$ and $|f_i(a) - a_i^*| = |f_i(a) - 1| = |1 + n_i(a)| \times |a_i - 1| \leq |a_i - a_i^*|$.
3. $n_i(a^*) = 0$ and $n_i(a) \geq 0$: $|f_i(a) - a_i^*| = |(1 - n_i(a))a_i - a_i^*| = |a_i - a_i^* - n_i(a)a_i| \leq |a_i - a_i^*|$.
4. $n_i(a^*) = 0$ and $n_i(a) \leq 0$: $|f_i(a) - a_i^*| = |(1 + n_i(a))a_i - n_i(a) - a_i^*| = |a_i - a_i^* - n_i(a)(1 - a_i)| \leq |a_i - a_i^*|$.

Therefore $\forall a \in N^\epsilon(a^*), |f_i(a) - a_i^*| \leq |a_i - a_i^*|$ and $|f(a) - a^*| \leq |a - a^*|$. According to Lemma 2, a^* is stable. Furthermore, let $V(a) = |a - A^*|$ be defined on a neighborhood of A^*, V is a Liapunov function for A^* and SM. According to Theorem 2, A^* is an attractor. □

4.2 Global consistency

Unconstrained optimization methods can also be used to solve a set of equations $g_i(x) = 0, i = 1..n$, by letting $\mathcal{E}_g(x) = \Sigma_{i=1..n} w_i g_i^2(x)$ where $w_i > 0$ and $\Sigma_i w_i = 1$. If a constraint solver CS solves $\min \mathcal{E}_g(x)$, CS solves $g(x) = 0$. Inequality constraints can be transformed into equality constraints. There are two approaches. Let $g_i(x) \leq 0$ be an inequality constraint, the equivalent equality constraint is (i) $max(0, g_i(x)) = 0$ or (ii) $g_i(x) + z^2 = 0$ where z is introduced as an extra variable. Similarly, the schema model can be used to solve a set of constraints with finite domains, by assigning each possible value a unit and each constraint between two values a weight.

However, in many cases it is more efficient to solve a set of (in)equality constraints directly. Moreover, a method for solving a set of equality constraints can also be used to solve an unconstrainted optimization problem, since x^* is a local extremum of $\mathcal{E}$ implies $\frac{\partial \mathcal{E}}{\partial x}(x^*) = 0$. Similarly, the problem of finite domain constraint satisfaction can be solved directly on constraint nets.

Here we first discuss the projection method (PM) for solving (in)equality constraints, and then study the method for solving global consistency of finite domain constraints (FM).

4.2.1 Projection method

A *projection* of a point x to a set R in a metric space $\langle X, d\rangle$ is a point $P_R(x) \in R$ such that $d(x, P_R(x)) = d(x, R)$. Projections in the n-dimensional Euclidean space $\langle \mathcal{R}^n, d_n\rangle$ share the following properties.

Lemma 4 [8] *Let $R \subset \mathcal{R}^n$ be closed and convex*[4]. *The projection $P_R(x)$ of x to R exists and is unique for every x, and $(x - P_R(x))^T(y - P_R(x)) \leq 0$ for any $y \in R$.*

Suppose we are given a system of convex and closed sets, X_i for $i = 1..m$. The problem is to find $x^* \in \cap_i X_i$. Let $P(x) = P_{X_l}(x)$ be a projection of x to a least satisfied set X_l, i.e. $d(x, X_l) = \max_i d(x, X_i)$. The projection method [8] for this problem defines a state transition system $\langle \mathcal{R}^n, f\rangle$ where $f(x) = x + \lambda(P(x) - x)$ for $0 < \lambda < 2$.

The following theorem is derived from a similar one in [8], however, the proof given here is simplified by the use of Liapunov functions.

Theorem 6 *Let PM be a constraint net representing the projection method. PM solves $X^* = \cap_i X_i$ globally if all the X_i's are convex.*

Proof: First of all, it is easy to see that if x^* is a solution, then $x^* = f(x^*)$, i.e. x^* is an equilibrium. Moreover, we can prove that $|f(x) - x^*| \leq |x - x^*|$ for any x as follows.

$$\begin{aligned}
|f(x) - x^*|^2 &= |x + \lambda(P(x) - x) - x^*|^2 \\
&= |x - x^*|^2 + \lambda^2|P(x) - x|^2 + 2\lambda(x - x^*)^T(P(x) - x) \\
&= |x - x^*|^2 + (\lambda^2 - 2\lambda)|P(x) - x|^2 + 2\lambda(P(x) - x)^T(P(x) - x^*) \\
&\leq |x - x^*|^2 - \lambda(2 - \lambda)|P(x) - x|^2 \quad \text{according to Lemma 4} \\
&\leq |x - x^*|^2 \quad \text{since } 0 < \lambda < 2.
\end{aligned}$$

According to Lemma 2, x^* is stable.

Then, we can prove that X^* is an attractor in the large. Let $V(x) = |x - X^*|$ on $\mathcal{R}^n$. $V(x)$ is a Liapunov function on $\mathcal{R}^n$ since $V(f(x)) \leq V(x)$ for any x. Moreover $V(f(x)) < V(x)$ for any $x \notin X^*$ since for any $x^* \in X^*$ and $x \notin X^*$, $|f(x) - x^*| < |x - x^*|$. In addition, we can prove that the process defined by PM satisfies the condition in Theorem 2, since $\lim_{n\to\infty} x^n = x^*$ implies $\lim_{n\to\infty} |x^{n+1} - x^n| = 0$. Therefore $\lim_{n\to\infty} |P(x^n) - x^n| = 0$ and $P(x^*) = x^*$, so that x^* is an equilibrium. According to Theorem 2, X^* is an attractor in the large. □

The projection method can be used to solve a set of inequality constraints, i.e. $X_i = \{x | g_i(x) \leq 0\}$ for convex function g_i. Linear functions are convex. Therefore the projection method can be applied to a set of linear inequalities $Ax \leq b$, where $x = \langle x_1, \ldots, x_n\rangle \in \mathcal{R}^n$. Let A_i be the ith row of A. The projection of a point x to a half space $A_i x - b_i \leq 0$ is defined as:

$$P_i(x) = \begin{cases} x & \text{if } A_i x - b_i \leq 0 \\ x - cA_i^T & \text{otherwise} \end{cases}$$

where $c = (A_i x - b_i)/|A_i^T|^2$. This reduces to the method described in [1]. Without any modification, this method can be also applied to a set of linear equalities, by simply replacing each linear equality $g_i(x) = 0$ with two linear inequalities: $g_i(x) \leq 0$ and $-g_i(x) \leq 0$.

There are various ways to modify this method for faster convergence. For instance, [3] gives a simultaneous projection method in which $f(x) = x + \lambda\Sigma_{i\in I} w_i(P_i(x) - x)$ where I is an index set of violated constraints, $w_i > 0$ and $\Sigma_{i\in I} w_i = 1$. [21] gives a method in which $f(x) = x + \lambda(P_S(x) - x)$ where $S = \{x | \Sigma_{i\in I} w_i g_i(x) \leq 0\}$. Furthermore, for a large set of inequalities, the problem can be decomposed into a set of K subproblems with f_k corresponding to the transition function of the kth subproblem. The whole problem can be solved by combining the results of $\{f_1, \ldots, f_K\}$.

[4] A set R in n-dimensional Euclidean space is convex iff for any $\lambda \in (0,1), x, y \in R$ implies $\lambda x + (1 - \lambda)y \in R$. Clearly if g is a convex function, $\{x | g(x) \leq 0\}$ is a convex set.

4.2.2 Finite constraint satisfaction

Many problems can be formalized as finite constraint satisfaction problems (FCSPs), which can be represented by constraint networks [24]. Formally, a *constraint network* C is a quadruple $\langle V, dom, A, con \rangle$ where

- V is a set of variables, $\{v_1, v_2, \ldots, v_N\}$,
- associated with each variable v_i is a finite domain $d_i = dom(v_i)$,
- A is a set of arcs, $\{a_1, a_2, \ldots, a_n\}$,
- associated with each arc a_i is a constraint $con(a_i) = r_i(R_i)$ where $R_i \subseteq V$ is a relation scheme and r_i is a set of relation tuples on R_i.

The *solution set* for the constraint network C is the join of all the relations, $sol(C) = r_1 \bowtie \ldots \bowtie r_n$.

An FCSP can be solved using the schema model (SM) by assigning each possible value in the finite domain of a variable a unit. The units of two values from the same variable are against each other; the units of two values from different variables support each other if they are consistent. However, SM does not solve an FCSP globally.

An FCSP can be solved directly using various methods [4, 6, 11, 12, 14]. Let $Scheme(C) = \{R_1, \ldots, R_n\}$ be the scheme of a constraint network C. The *solution of a constraint network* C is a network C', with $sol(C) = sol(C')$, $Scheme(C) = Scheme(C')$, and $r'_i = \Pi_{R_i}(sol(C'))$ where Π_{R_i} is a projection operator. Such a solution network is called a *minimal* network [14]. Here we present a relaxation method (FM) which finds the solution network of a constraint network with an acyclic scheme. This kind of method has been studied by many researchers, for instance, [7, 17, 24]. We examine the property of the method within the framework of dynamic systems.

Let $\mathcal{C}$ be the set of constraint networks with the same scheme and solution set. We define a state transition system $\langle \mathcal{C}, f \rangle$ where $f = \{f_i\}_{a_i \in A}$ with $f_i(r_i) = \bigcap_{\{j | R_i \cap R_j \neq \emptyset\}} \Pi_{R_i}(r_i \bowtie r_j)$.

Theorem 7 *Let FM be a constraint net representing a state transition system $\langle \mathcal{C}, f \rangle$. FM finds the solution network globally in $\mathcal{C}$ if the scheme of $\mathcal{C}$ is acyclic.*

Proof: First of all, it is clear that a solution network C^* is an equilibrium of the state transition system. Now let us define a metric on the set $\mathcal{C}$. Given a relation scheme R, the distance between two relation tuples can be defined as $d_R(r_1, r_2) = |(r_1 - r_2) \cup (r_2 - r_1)|$ where $|r|$ denotes the number of relation tuples. The distance between two constraint networks in $\mathcal{C}$ can be defined as $d(C_1, C_2) = \sqrt{\Sigma_{Scheme(\mathcal{C})} d_R^2(r_1, r_2)}$. Let us define a function L on $\mathcal{C}$ as: $L(C) = \sqrt{\Sigma_{Scheme(\mathcal{C})} |r|^2}$. L is a Liapunov function for the solution network C^* and $\langle \mathcal{C}, f \rangle$ since (1) $L(C_1) \leq L(C_2)$ iff $d(C_1, C^*) \leq d(C_2, C^*)$ and (2) $L(f(C)) \leq L(C)$ for any $C \in \mathcal{C}$. Therefore C^* is a stable equilibrium. Finally, we can prove that if the scheme of $\mathcal{C}$ is acyclic, C^* is an asymptotically stable equilibrium. For an acyclic network, an equilibrium implies a minimal network [24] and clearly if $C \neq C^*$, $L(f(C)) < L(C)$. According to Theorem 2, C^* is asymptotically stable.□

5 Embedded Constraint Solvers and Implementation Issues

In this section, we consider two variations of constraint solvers. The first corresponds to open constraint nets, for designing embedded control systems. The second corresponds to constraint nets with latency.

5.1 Embedded constraint solvers

One of the important applications of constraint solvers is the design of robot control systems [15]. There are two kinds of embedded constraint solvers for this application. First, a constraint solver is coupled to a dynamic environment. Second, a constraint solver is coupled to the plant of a robot. In both cases, the embedded constraint solver is part of the robot controller. The combination of these two embeddings will occur in real applications.

An embedded constraint solver coupled to an environment (resp. a plant) is an open constraint net $CN(I, O)$, where the set of input locations I act as sensors of the environment (resp. the plant). Constraints

are relations on input and output values. A constraint net is an *embedded constraint solver* for the set of constraints C and the environment (resp. the plant) iff the composition of the constraint net and the environment (resp. the plant) solves C.

Consider the case of designing a tracking system S which chases a target T. Let x be the position of S and x_d be the position of T, the constraint to be satisfied is $|x - x_d| = 0$. Suppose we design a tracking system with the following law: $\frac{dx}{dt} = -k(x - x_d)$ where x_d is an input trace[5]. However, this system is not an embedded constraint solver for $|x - x_d| = 0$ if $|\frac{dx_d}{dt}| > 0$. A correct design is $\frac{dx}{dt} = \frac{dx_d}{dt} - k(x - x_d)$. To see why this is the right design, we define a Liapunov function $V(x, x_d) = \frac{1}{2}|x - x_d|^2$ and observe that $\frac{dV}{dt} = (x - x_d)(\frac{dx}{dt} - \frac{dx_d}{dt}) \leq 0$. In reality, both x_d and $\frac{dx_d}{dt}$ can be inaccurate. However, the system is robust with respect to the inaccuracy.

5.2 Implementation issues

Constraint solvers (or embedded constraint solvers) can be implemented as analog or digital circuits, or as programs in multiprocessor environments. For a discrete constraint solver, the efficiency can be characterized by the convergence rate of the method and the computation cost of the transition function. Constraint nets are inherently parallel, while sequential computation can be considered as a special case. Clearly, for discrete constraint solvers, except those embedded in dynamic environments, the computation time will not affect the dynamic behaviors.

However, for a continuous constraint solver, latencies in the circuit may change the dynamic behavior of the constraint solver totally. Consider a simple example: $\frac{dx}{dt} = -kx$ with $k > 0$ solves $x = 0$ globally. However, if there is a latency δ in the wires, the actual equation becomes $\frac{dx}{dt} = -kx(t - \delta)$. In this case, $x = 0$ is still an equilibrium, but it may not be an asymptotically stable equilibrium. In fact, if $\delta k > 2$, it is unstable at $x = 0$. Therefore, it is important at the design stage to model the possible latencies and to choose the right value for k.

6 Conclusion

We have presented a unitary model of constraint satisfaction as a dynamic process. Various constraint methods and their dynamic properties have been studied, and their applications to control system design are examined. The Constraint Net model serves as a useful abstract target machine for constraint programming languages, providing both semantics and pragmatics.

Acknowledgements: We wish to thank Uri Ascher, Peter Lawrence, Dinesh Pai, Nick Pippenger and Runping Qi for valuable discussions and suggestions. This research was supported by the Natural Sciences and Engineering Research Council and the Institute for Robotics and Intelligent Systems.

References

[1] S. Agmon. The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, 6:382–392, 1954.

[2] A. Aiba, K. Sakai, Y. Sato, and D.J. Hawley. Constraint logic programming language cal. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 263 -276, 1988.

[3] Y. Censor and T. Elfving. New method for linear inequalities. *Linear Algebra and Its Applications*, 42:199–211, 1982.

[4] R. Dechter. Constraint networks. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285 - 293. Wiley, N.Y., 1992.

[5] In practice, $x - x_d$ would be sensed.

[5] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 693 - 702, 1988.

[6] E. C. Freuder. Complexity of k-tree structured constraint satisfaction problems. In *Proceeding of AAAI-90*, 1990.

[7] E. C. Freuder. Completable representations of constraint satisfaction problems. In *KR-91*, pages 186 - 195, 1991.

[8] L.G. Gubin, B.T. Polyak, and E.V. Raik. The method of projections for finding the common point of convex sets. *U.S.S.R. Computational Mathematics and Mathematical Physics*, pages 1-24, 1967.

[9] J. Jaffar and J.L. Lassez. Constraint logic programming. In *ACM Principles of Programming Languages*, pages 111 - 119, 1987.

[10] D.G. Luenberger. *Introduction to Dynamic Systems: Theory, Models and Applications.* John Wiley & Sons, 1979.

[11] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 276 - 285. Wiley, N.Y., 1992.

[12] A.K. Mackworth. The logic of constraint satisfaction. *Artificial Intelligence*, 58:3-20. 1992.

[13] M. D. Mesarovic and Y. Takahara. *General Systems Theory: Mathematical Foundations.* Academic Press, 1975.

[14] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7:95-132, 1974.

[15] D. K. Pai. Least constraint: A framework for the control of complex mechanical systems. In *Proceedings of American Control Conference*, pages 426 - 432, Boston, 1991.

[16] J. Platt. Constraint methods for neural networks and computer graphics. Technical Report Caltech-CS-TR-89-07, Department of Computer Science, California Institute of Technology, 1989.

[17] F. Rossi and U. Montanari. Exact solution in linear time of networks of constraints using perfect relaxation. In *Proceedings First Int. Principles of Knowledge Representation and Reasoning, Toronto, Ontario, Canada*, pages 394-399, May 1989.

[18] D. E. Rumelhart and J. L. McClelland, editors. *Parallel Distributed Processing — Exploration in the Microstructure of Cognition.* MIT Press, 1986.

[19] J. T. Sandfur. *Discrete Dynamical Systems: Theory and Applications.* Clarendon Press, 1990.

[20] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. Technical Report SSL-90-86, Palo Alto Research Center, 1990.

[21] K. Yang and K.G. Murty. New iterative methods for linear inequalities. Unpublished.

[22] Y. Zhang and A. K. Mackworth. Constraint nets: A semantic model of real-time embedded systems. Technical Report 92-10, Department of Computer Science, University of British Columbia, 1992.

[23] Y. Zhang and A. K. Mackworth. Will the robot do the right thing? Technical Report 92-31, Department of Computer Science, University of British Columbia, 1992.

[24] Y. Zhang and A. K. Mackworth. Parallel and distributed constraint satisfaction: Complexity, algorithms and experiments. In Laveen N. Kanal, editor, *Parallel Processing for Artificial Intelligence.* Elsevier/North Holland, 1993. to appear.

A Constraint Based Scientific Programming Language

Richard Zippel
Cornell University
Ithaca, NY 14853
rz@cs.cornell.edu

February 15, 1993

Scientific programs tend to be quite large and complex, and their creation is quite error prone. We have been pursuing a program transformation approach to the creation of scientific programs, where the transforms can be conventional compiler optimizations like loop unrolling, strength reduction and common subexpression elimination, and more mathematical transformation like applying Newton's method to a coupled system of algebraic equations or a Runge-Kutta method to a system of ordinary or partial differential equations.

The language to which these transforms is applied contains, by necessity, a combination of conventional programming constructs and constructs from continuous mathematics such as differential equations, integrals and function spaces. We call this language *SPL*. A very appropriate way to combine these different constructs is to begin with a conventional programming language, but one where the type system is extended to include types like continuous functions and further extend it to use constraints to represent the algebraic and differential equations of the scientific problem.

The constraints are the mechanism by which the physical problem is modeled. Two additional mechanisms are introduced in the constraint system to provide sufficient physical modeling flexibility: (1) the constraints themselves are scoped in much the same manner as variable bindings, and (2) predicates are provided to control when the constraints are applicable. The first mechanism permits standard code walking and analysis tools to be used on the the scientific programs, while the second allows us to model changes in the physical model that occur when, say, two mechanical bodies come into contact, or in combustion, the fuel is completely consumed.

The constraint programs that we produce are not executable in the sense that conventional constraint networks are. Rather they are interpreted as a specification of what needs to be computed. By sequences of mathematical transformations we are able to compile these high level scientific constraint programs into executable code.

1 SPL Design Goals

We note that there are two different communities that engage in program transformations, but which use rather different languages and techniques. On the one hand there are those involved in compilers, program transformations and partial evaluation in the computer science community. This community, or communities, is interested in source to source transformations of programming languages such as loop unrolling, common subexpression elimination and strength reduction. It is this community that one usually thinks of when the phrase "program transformations" is mentioned.

On the other hand there is the numerical analysis/scientific computing community, whose transformations are on mathematical objects like differential equations. Typical of their transformations

are the following.

- Approximation

$$\sin x \Longrightarrow x - \frac{x^3}{6} + \frac{x^5}{120}$$

- Horner's rule

$$x - \frac{x^3}{6} + \frac{x^5}{120} \Longrightarrow x \cdot \left(1 - x^2 \left(\frac{1}{6} - \frac{x^2}{120}\right)\right)$$

- discretization

$$\frac{dx}{dt} = f(x) \Longrightarrow \frac{x^{n+1} - x^n}{\Delta t} = f(x^{n+1})$$

Although the numerical analysis/scientific computing community usually does not think of these transformations as operating on programs, notice that the discretizations of ordinary differential equations convert a single equation into a sequence of assignments.

One of the goals of the on-going design of SPL is that be able to support both types of program transformations and provide a basis for cooperation between these two communities. As a consequence, SPL must be able to represent the constructs of conventional programming languages: variables, arrays, structures, loops, recursions, etc. and it must must support the constructs of the scientific computing community: continuous functions, differential equations, function spaces, etc.

In addition, we feel it important that it be possible to specify scientific computations in SPL in a way that does not bias how the computation is actually performed. Thus we want to be able to say that a certain function is the solution of a differential equation, with out specifying the numerical method to be used to compute the function. This means that by specifying the proper program transformations on will be able to produce programs that use computational methods that are appropriate for different architectures.

2 SPL Examples

This section gives a couple of examples of the SPL language and a few of the transformations that are being developed. For expository purposes we have used an infix syntax for SPL programs although we use a parenthesized Lisp syntax in our implementations. The examples given here are quite simple, and our main purpose is to illustrate the use of the constraints and how they are transformed in to code.

2.1 System of Algebraic Equations

Assume we wish to know the real values of y such that there is exists real numbers x satisfying

$$x^2 + y^3 = 5,$$

$$xy + y^2 = 3.$$

The following program specifies this computation.

```
declare x, y ∈ ℝ;
  [T ⟹ {x^2 + y^3 = 5, xy + y^2 = 3};
    print(y);
    ]
```

The first statement of the program indicates that x and y are only allowed to take on real values. The square brackets delimit the scope of a set of constraints. The constraints themselves are qualified and are written with the syntax:

$$\langle \textit{predicate} \rangle \Longrightarrow \langle \textit{set of mathematical equations} \rangle$$

The set of constraints indicated is to be enforced only within the square brackets and only when the predicate is true.

In the program above, the constraints always take effect since the predicate is T (= true). The body (just a print statement in this case) is executed for every (x, y) pair that satisfies the system of equations, not just once. (In this case, the three real values that y can take on would be printed.)

There are two things to note about the program above. First, it is a mixture of mathematical constructs and the programming constructs. The print statement is a programming construct while the constraint equations are mathematical and do not themselves contain computational content. Second, the program above is not executable in the form given. Some transformations need to be applied to transform the constraints to a more conventional programming paradigm.

One approach would be as follows. Since the body of the program does not involve x, it makes sense to try to eliminate it from the constraint equations. If the "lexical Gröbner basis transform" is applied to constraint "$T \Longrightarrow \{x^2 + y^3 = 5, xy + y^2 = 3\}$" we end up with:

```
declare x, y ∈ ℝ;
  [ T ⟹ {y^5 + y^4 − 11y^2 + 9 = 0, x + y^4/3 + y^3/3 + 8y/3 = 0};
    print(y);
    ]
```

Now since the constraint in x is linear, any real value of y will give a real value of y. So we can drop the second constraint and all reference to x.

```
declare y ∈ ℝ;
  [ T ⟹ {y^5 + y^4 − 11y^2 + 9 = 0};
    print(y);
    ]
```

At this point we can apply the "Newton's method transformation"

```
declare y ∈ ℝ;
  loop y ← FindInitialSoln({y^5 + y^4 − 11y^2 + 9 = 0}) do {;
    loop y ← y − (y^5+y^4−11y^2+9)/(5y^4+4y^3−11y);
         until convergence;
    print(y);
    }
```

The phrase `until convergence` covers a number of details will be discussed in the full paper.

2.2 *RC* Circuit

Assume we have a simple *RC* circuit driven by a voltage of $V_i(and)$. The device equations for the resistor and capacitor give the constraints

$$V_i(t) - V(t) = R \cdot I(t),$$

$$I(t) = C\frac{dV(t)}{dt}.$$

In SPL, we can write this as

```
declare V, I ∈ C^∞(ℝ^+ → ℝ);
  [t ∈ ℝ^+ ⟹ {V_i(t) − V(t) = R · I(t), I(t) = C dV(t)/dt};
    print(V(t));
    ]
```

Again, the body only involves V, so we can eliminate I from the equations to give

```
declare V ∈ C^∞(ℝ^+ → ℝ);
  [t ∈ ℝ^+ ⟹ {V_i(t) − V(t) = RC dV(t)/dt};
    print(V(t));
    ]
```

At this point we have a constraint on continuous functions that involves a derivative. The usual constraint propagation techniques do not work in this case. In the work of Stallman, Steele and Sussman on circuits, they took the approach of writing the constraints in the frequency domain, where the differential equation becomes a linear equation. In our context that corresponds to taking the Fourier transform of the above program:

```
declare v ∈ ℱ · C^∞(ℝ^+ → ℝ);
  [s ∈ ℂ ⟹ {ℱ{V_i(t)} − v(s) = sRCv(s)};
    print(ℱ^{-1}{v(s)});
    ]
```

In this case however, we would like to generate a standard transient analysis and thus want to numerically integrate the differential equation. This is done by discretizing time using the backward Euler formula.

```
declare V[] ∈ C^∞(→ ℝ);
  [n ∈ {0, 1, 2, ...} ⟹ {V_i(n Δt) − V[n + 1] = RC (V[n+1] − V[n])/Δt};
    print(V[]);
    ]
```

At this point the continuous function $V(t)$ has been discretized into an array of numbers. Thus the i^{th} element of the array $V[]$ should be $V[i] = V(i\,\Delta t)$.

Isolating $V[n+1]$ and converting to a loop we have

```
declare V[] ∈ C^∞(→ ℝ);
  [n ∈ {0, 1, 2, ...} ⟹ {V[n + 1] = (1 + Δt/RC)^{-1} [V[n] + Δt/RC V_i(n Δt))]};
    print(V[]);
    ]
```

At this point we could convert the program to one that is executable by converting the constraint into an assignment and loop over values of n. Since no initial conditions have been provided, there isn't much point in this.

2.3 Burgers' Equation Example

In this subsection we consider a partial differential equation, Burgers' equations:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \frac{\partial^2 u}{\partial x^2},$$

with periodic boundary conditions: $u(x + L, t) = u(x, t)$.

Now the initial program is a bit more complex:

```
D ← Per(0, L);
declare u ∈ C^∞(D × R^+ → R);
  [(x ∈ D) ∧ (t ∈ [0, T]) ⟹ {du/dt + u du/dx = d^2u/dx^2};
    [t = 0 ⟹ u(x, t) = u_0(x);
      print (u(x, t));
    ]
  ]
```

D is the interval 0 to L made periodic. The declaration indicates that u is an infinitely differentiable function from the direct sum of D and the positive real numbers to the real numbers. The outer constraint is constrains u to satisfy Burger's equation when x is in D and when t is in the time interval of interest. The inner constraint provides the initial conditions. This approach (and the nesting of the constraints) allows us to state problems with different initial conditions in a single program.

If time is discretized in uniform steps of length Δt, using the forward Euler method, we get the following program

```
D ← Per(0, L);
declare u[] ∈ C^∞(D → R);
  loop for n ∈ {0, 1, 2, ..., T/Δt} do {
    [ x ∈ D ⟹ {(u[n+1] − u[n])/Δt + u[n] du[n]/dx = d^2u[n]/dx^2};
      [ n = 0 ⟹ u[n](x) = u_0(x);
        print (u[n](x));
      ]
    ]
  }
```

Notice that after the time discretization, unlike the simple circuit problem, the constraint is still a differential equation not an algebraic equation. Further discretizations in space are needed to convert this program into something that is close to executable. Nonetheless, at this point we could apply compiler optimizations and parallelizations to the program (although in this case there is not much to optimize).

3 Conclusions

We have presented some examples of a scientific programming language that combines constraints with conventional programming tools. The constraints are used to represent the mathematical aspects of the program, such as the differential or algebraic equations that functions must satisfy.

Programs in this language are converted into executable form using a library of program transformations, which spans compiler optimizations, parallelizations and mathematical discretizations.

Author Index

Aït-Kaci, H., 1

Barahona, P., 201
Benhamou, F., 239
Boortz, K., 109
Brand, P., 109
Brodsky, A., 6
Brown Jr., A. L., 14

Carlson, B., 109
Codognet, P., 201
Cruz, I. F., 24

Danielsson, B., 109
Dasiewicz, P., 279
Delcher, A. L., 149
Donikian, S., 36
Dubé, T., 46
Duby, C. K., 211

Fages, F., 53
Fernando, T., 62
Frühwirth, T., 82
Franzén, T., 109
Freuder, E. C., 72

Gao, H., 92
Gleicher, M., 100

Hégron, G., 36
Hanschke, P., 82
Haridi, S., 109
Hubbe, P. D., 72

Imbert, J., 119

Jégou, P., 132
Janson, S., 109
Johnson, M., 140

Kasif, S., 149
Keirouz, W. T., 156
Keisu, T., 109
Kirchner, C., 166
Kirchner, H., 166
Kramer, G. A., 156
Kuper, G. M., 176

Lapalme, G., 184
Lassez, C., 6

Mackworth, A. K., 303
Major, F., 184
Mantha, S., 14
McAloon, K., 189
Menezes, F., 201
Meyers, S., 211
Michaylov, S., 221
Montanari, U., 230
Montelius, J., 109

Older, W. J., 239

Pabon, J., 156
Pai, D. K., 250
Park, K., 294
Pfenning, F., 221
Podelski, A., 1

Reiss, S. P., 211
Rossi, F., 230
Rounds, W. C., 258

Sahlin, D., 109
Sannella, M., 268
Savor, T., 279
Sjöland, T., 109
Smith, D. R., 288
Smith, T. R., 294

Tretkoff, C., 189
Turcotte, M., 184

Vittek, M., 166

Wakayama, T., 14
Ward, A. C., 299
Warren, D. S., 92

Yap, C., 46

Zhang, G., 258
Zhang, Y., 303
Zippel, R., 313