

NAVAL POSTGRADUATE SCHOOL Monterey, California



19990707 090

**Proceedings of
The 1998 ARO/ONR/NSF/DARPA Monterey Workshop on
Engineering Automation for Computer Based Systems**

By

Luqi

April 1999

Approved for public release; distribution is unlimited.

Prepared for: Naval Postgraduate School
Monterey, Ca 93943-5000

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

RADM Robert C. Chaplin
Superintendent

Richard S. Elster
Provost

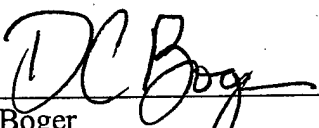
This report was prepared for Naval Postgraduate School
and funded in part by ARO/ONR/NSF/DARPA

This report was prepared by:



Luqi
Professor, Computer Science

Reviewed by:



Dan Boger
Chairman, Computer Science

Released by:



D. W. Netzer
Associate Provost and
Dean of Research

REPORT DOCUMENTATION PAGE			Form approved OMB No 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE April 1999		3. REPORT TYPE AND DATES COVERED Technical Report
4. TITLE AND SUBTITLE <i>Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems</i>			5. FUNDING NUMBERS <i>ARO (MIPR8GNPSAR042) NSF (CCR-9813820) ONR (N0001499WR20019) SPAWAR (N6600198WR00438)</i>	
6. AUTHOR(S) <i>Luqi</i>				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) <i>Software Engineering Group, Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943</i>			8. PERFORMING ORGANIZATION REPORT NUMBER <i>NPS-CS-99-002</i>	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) <i>ARO/ONR/NSF/DARPA</i>			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES <i>The views expressed in this report are those of the authors and do not reflect the official policy or position of the Department of Defence or the U.S. Government.</i>				
12a. DISTRIBUTION/AVAILABILITY STATEMENT <i>Approved for public release; distribution is unlimited.</i>			12b. DISTRIBUTION CODE <i>A</i>	
13. ABSTRACT (Maximum 200 words.) <i>The "Engineering Automation for Computer Based Systems" Workshop is the 6th in a series of Software Engineering workshops for formulating and advancing software engineering models and techniques, with the fundamental theme of increasing the practical impact of formal methods. Previous workshops have been devoted to "Real-time & Concurrent Systems", "Software Merging and Slicing", "Software Evolution", "Software Architecture", and "Requirements Targeting Software". A major goal for this series of workshops is to help focus the software engineering community on issues that are vital to improving the state of software engineering practice. This focus promotes consistency among diverse research directions that address different aspects of the same problem to facilitate future integration efforts.</i> <i>The workshop represents a bridge between industry and academia. The material in these proceedings presents a balanced view of academic and industrial developments. Formalization is fundamental to the development of software engineering as an engineering discipline. The critical importance of formal models and formal methods is painfully clear when one considers the escalating demands for larger, more complex, reliable software systems.</i>				
14. SUBJECT TERMS <i>Real-time, Concurrent Systems, Software Merging and Slicing, Software Evolution, Software Architecture, and Requirements Targeting Software</i>			15. NUMBER OF PAGES <i>163</i>	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT <i>Unclassified</i>	18. SECURITY CLASSIFICATION OF THIS PAGE <i>Unclassified</i>	19. SECURITY CLASSIFICATION OF ABSTRACT <i>Unclassified</i>		20. LIMITATION OF ABSTRACT <i>UL</i>

Proceedings of the
1998 Monterey Workshop on
Engineering Automation for
Computer Based Systems

Carmel, California
23-26 October, 1998

Sponsored by

ARO, ONR, NSF, DARPA, & NPS

**Proceedings of the
1998 ARO/ONR/NSF/DARPA Monterey Workshop on
Engineering Automation for Computer Based Systems**

**Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5118**

Abstract

The "Engineering Automation for Computer Based Systems" Workshop is the 6th in a series of Software Engineering workshops for formulating and advancing software engineering models and techniques, with the fundamental theme of increasing the practical impact of formal methods. Previous workshops have been devoted to "Real-time & Concurrent Systems", "Software Merging and Slicing", "Software Evolution", "Software Architecture", and "Requirements Targeting Software". A major goal for this series of workshops is to help focus the software engineering community on issues that are vital to improving the state of software engineering practice. This focus promotes consistency among diverse research directions that address different aspects of the same problem to facilitate future integration efforts.

The workshop represents a bridge between industry and academia. The material in these proceedings presents a balanced view of academic and industrial developments. Formalization is fundamental to the development of software engineering as an engineering discipline. The critical importance of formal models and formal methods is painfully clear when one considers the escalating demands for larger, more complex, reliable software systems.

Workshop Chairs

Luqi and Manfred Broy

Advisory Committee

**David Hislop
Frank Anger
Valdis Berzins
Daniel Cooke
Michael DaBose**

Program Committee

**Manfred Broy, Technical University of Munich
Insup Lee, University of Pennsylvania
Luqi, Naval Postgraduate School
Zohar Manna, Stanford University
Alan Shaw, University of Washington**

Organization Committee

**Naval Postgraduate School
Man-Tak Shing, Jiang Guo, Beeyen Wong, Mickey Harn**

Acknowledgement

The organizers of this workshop would like to thank the sponsors of the workshop: Army Research Office (ARO), Office of Naval Research (ONR), National Science Foundation (NSF), Defense Advanced Research Projects Agency (DARPA), and NPS (Naval Postgraduate School).

List of Attendees

Anger, Frank National Science Foundation	Gelfond, Michael Univ. of Texas at El Paso	Polak, Wolfgang Consultant
Astesiano, Egidio Univ. Di Genova	Green, Cordell KESTREL	Pratt, Vaughan Stanford Univ.
Berry, Dan Technion/Univ. of Waterloo	Gill, Helen Defense Advanced Research Projects Agency	Ray, Bill SPAWAR
Berzins, Valdis Naval Postgraduate School	Guo, Jiang Naval Postgraduate School	Robertson, Dave University of Edinburgh
Bjorner, Nikolaj Stanford University	Harn, Meng-chyi Naval Postgraduate School	Shatz, Sol Univ. Illinois, Chicago
Broy, Manfred Tech. Univ. of Munich	Hislop, David Army Research Office	Shaw, Alan Washington Univ.
Carver, Doris Louisiana State University	Iyer, Purush North Carolina State University	Shen, Lydia SPAWAR
Cleaveland, Rance SUNY, Stony Brook	Kraemer, Bernd Fern Univ. Hagen	Shing, Man-Tak Naval Postgraduate School
Cooke, Dan Univ. of Texas at El Paso	Lange, Doug SPAWAR	Uribe, Tomas Stanford University
Dabose, Mike SPAWAR	Lee, Insup University of Penn	Waldinger, Rich SRI
Delgado, Ref Defense Advanced Research Projects Agency	Levitt, Raymond Stanford University	Wegner, Peter Brown University
Devanbu, Premkumar University of California, Davis	Lipton, James Wesleyan University	Wirsing, Martin Univ. Muenchen
Drumond, John SPAWAR	Liu, Junbo KESTREL	Yehudai, Amiram Tel Aviv Univ.
Ehrich, H.D. Univ. Braunschweig	Luqi Naval Postgraduate School	Zhang, Du Cal State University of Sacramento
Evans, John SPAWAR	Mislove, Mike Tulane University	Zhao, Feng Xerox PARC

Table of Contents

Software Engineering to our Planning Horizon Luqi, Manfres Broy	1
Engineering Automation for Computer Based Systems Luqi	3
Formal Methods: The Very Idea*, Some Thoughts About Why They Work When They Work Daniel M. Berry	9
Light Weight Inference for Automation Efficiency V. Berzins	19
Reactive Verification with Queues Nikolaj S. Bjorner	30
Generic Tools for Verifying Concurrent Systems Rance Cleaveland, Steven T. Sims	38
Automatic Concurrency in SequenceL Daniel E. Cooke, Vladik Kreinovich	47
On Methodology of Representing Knowledge in Dynamic Domains Michael Gelfond, Richard Watson	57
The Role of Observations in Probabilistic Open Systems Murali Narasimha, Rance Cleaveland, Purushothaman Iyer	67
Deductive Model Checking and Abstraction Zohar Manna, Henny B. Sipma, and Tomas E. Uribe	77
Formal Methods in Practice Wolfgang Polak	86
Formalizing and Executing Message Sequence Charts via Timed Rewriting Piotr Kosiuczenko, Martin Wirsing	93

Real-Time Systems Development with MASS Vered Gafni, Yishai Feldman, Amiram Yehudai	105
Parametric Approach to the Specification and Analysis of Real-time System Designs based on ACSR-VP Hee-Hwan Kwak, Insup Lee, and Oleg Sokolsky	115
Automated Facts Generation from Raw Data: a Perspective from the Andes Project Du Zhang, Vo Lee, Joseph Friedel, Robert Keyser,	125
Pitfalls of Formality in Early System Design David Robertson	135
Automated Verification of Function Block Based Industrial Control Systems Norbert Volker, Bernd J. Kramer	142
The Story of Re-engineering of 350,000 Lines of FORTRAN Code M. Shing, Luqi, V. Berzins, M. Saluto, J. Williams, J. Guo, and B. Shultes	151
Data Transmission Over Fiber Optics Using High Performance Network Protocol John Drummond	161

Software Engineering to our Planning Horizon¹

Luqi and Manfred Broy

The Army Research Office, National Science Foundation, Office of Naval Research, and the Defense Advanced Research Projects Agency sponsored the 1998 Monterey Workshop on Engineering Automation for Computer Based Systems.

This workshop is the 6th in a series of international workshops with the general theme of increasing the practical impact of formal methods for software and systems engineering. The workshop took place in Carmel, California late 1998, hosted by the Naval Postgraduate School.

Since 1990, the previous workshops in the series focused on real-time and concurrent systems, software merging and slicing, software evolution, software architecture, and requirements targeting software. This workshop focused on engineering automation.

The objectives of the workshops are to encourage interaction between the research and engineering communities, exchange recent results, assess their significance and encourage transfer of relevant results to practice, communicate current problems in engineering practice to researchers, and help focus future research on directions that address pressing practical needs.

Over the past years, we have witnessed a slow but steady decrease in the gap between the theoretical and practical sides of the software engineering community. We hope that this trend will continue and will accelerate improvements in the state of software engineering practice and theory. Software problems have been quite visible to the public due to spectacular disasters in space missions or telephone black outs and are receiving increasing attention with the nearing Y2K deadline. It is a good time to demonstrate concrete improvements in our discipline.

The continued doubling of computing speed and memory capacity every 18 months implies that the only constancy for large distributed systems, technology, tactics and doctrine may well be the idea that change is always inevitable. The dynamic aspect of systems is not supported by current practice and is seldom emphasized in current research. Software evolution research is extremely important for achieving modifiable and dependable systems in the future. Improved methods for reengineering are also needed to bring legacy systems to the condition where they can benefit from improvements in software evolution technology.

Thirty years ago, when the term software engineering was coined, there was lack of theoretical foundation for many practical concepts in computing. That is no longer true. A solid body of foundational work is available now that addresses many challenging issues related to software and computing, including specification techniques for systems and data, logical calculi for concurrent, distributed, and real-time systems, logical concepts related to interactive systems, and formal models of programming language semantics with a variety of inference systems.

The challenge is to put these results to work, to develop theory that better supports engineering needs, and to improve practice. This will require cooperation and a concerted effort from both theoreticians and practitioners. We will need advances in education and

¹ This research was supported by ARO(MIPR8GNPSAR042), NSF(CCR-9813820), ONR(N0001499WR20019), SPAWAR(N6600198WR00438).

improvements in theoretical approaches to meet the demand of practical engineering for computer software. To be attractive to practitioners, formal methods, mathematical foundations and automated engineering tools need to provide return on investment. These approaches must be cost effective to successfully compete with other development methods, and the benefits they provide in terms of software quality must have sufficient economic value to justify investment in them.

These goals require some uncomfortable changes in the research community. Mathematical elegance is not enough for the success of an engineering theory: applicability, tractability, and ease of understanding are often more important in practice than logical completeness or conceptual elegance of the principles that guarantee the soundness of the methods. We must carefully separate the application of mathematics to demonstrate the soundness of a formal software model or to construct automated tools for engineers from the formal models that will be used *by engineers as design representations*.

The formal aspects of computing cannot be studied in isolation if we are to have practical impact. The different aspects of technical, educational, and management issues are so closely intertwined in software engineering practice that it is risky and ineffective to study and develop them in isolation if practical applicability is a prominent goal. This puts interdisciplinary requirements on researchers and lends importance to interactions between experts from different specialties, such as those promoted by this workshop.

We have collected some excellent papers for the workshop. These articles are written by internationally renowned contributors from both academia and industry that examine current best practices and propose strategies for improvement, as well as a summary of the high points of the discussions at the workshop.

The broadest range of expert opinion and views were represented. Members of the academic, government, military and commercial world came to share their vision, insight and concerns. By synthesizing the expertise of these communities we hope to gain significant insight into the problems and solutions. The discussions ranged beyond the narrow confines of software and mathematics, to address engineering of systems containing hardware and people as well as software, and related issues that include requirements elicitation, management, and engineering education. Discussions at the workshop addressed technical advances in mature areas, such as a new decision procedure for a queue data type and novel types of model checking, as well as ideas for new directions, such as lightweight inference and co-algebraic models for interactive systems. The workshop helped to reduce the gap between theory and practice, and to recharge the research community to address problems of immediate concern. Workshop attendees identified and discussed both the technologically dependent and technologically independent trends within the engineering automation of computer based systems for the near term and out to our planning horizon.

It is our pleasure to thank the workshop advisory, program and local arrangements committees, and the workshop sponsors, NSF, ONR, DARPA, and especially ARO, for their vision of a principled engineering solution for software and for their many-year tireless effort in supporting a series of workshops to bring everyone together.

Engineering Automation for Computer Based Systems¹

Luqi

Computer Science Department, Naval Postgraduate School, USA.

1. Introduction

Software development capabilities lag far behind society's demands for better, cheaper, more reliable software. Since the gap is so large, and widening, it is unlikely that "business as usual" will be able to meet this need. Engineering automation based on sound and scientific methods appears to be our best chance to close the gap.

This is the sixth in a series of workshops whose common goal is helping to increase the practical impact of formal methods in software development. These workshops have succeeded in gradually bringing the theoretical and practical sides of the software engineering community closer together, focusing them on fulfilling the promise of scientific improvement of software engineering practice. The progress made in this direction at this workshop was larger and more readily apparent than in previous years, giving us hope that the effort will eventually succeed.

The remainder of this paper is organized as follows. Section 2 restates the main premises of the workshop. Section 3 gives an overview of the papers. Section 4 summarizes some of the discussion at the workshop, and Section 5 presents some conclusions.

2. Premises of the Workshop

The main premises of the workshop are that mathematics and formal methods can help solve practical problems in the engineering of computer-based systems, and that engineering automation is a promising way to accomplish this.

We use a broad definition of "formal method." Webster's Dictionary says that *formal* means definite, orderly, and methodical; that *method* means a regular, orderly, and definite procedure; and that *model* is a preliminary representation that serves as a plan from which the final, usually larger, object is to be constructed. Thus, to be formal does not necessarily require the use of logic, or even of mathematics.

In computer science, the phrase *formal method* has taken on a narrower meaning, referring to the use of a formal notation to represent system models during program development. An even narrower sense refers to use of a formal logic to express system specifications, and proofs to check correctness of implementation code - i.e., that it satisfies the specification.

The broader definition of formal method is appropriate to this workshop because it fits the theme of engineering automation. Processes need to be definite, orderly, and methodical to be successfully and reliably automated. Thus, formalization of engineering processes in this broad sense is a prerequisite for engineering automation.

The narrower sense of formal method - checking whether or not the code satisfies a particular requirement specification in a formal logic - is inappropriate for this purpose, because of the well known fact that the majority of software defects are requirements errors (see the paper by Berry in this Proceedings). If the specification is wrong, we do not want code that satisfies the specification.

The broader interpretation of formal method opens the door to other approaches, such as requirements elicitation via prototyping and the automatic synthesis of correct code from requirements models

¹ This research was supported by ARO(MIPR8GNPSAR042), NSF(CCR-9813820), ONR(N0001499WR20019), SPAWAR(N6600198WR00438).

formulated via domain-specific notations. Note that a formal model is required to generate an executable version of a prototype, and practical prototyping requires extensive automation of the prototype design, analysis and implementation process. Such tools depend on extensive formalization of the processes involved. Similarly, the design of a domain-specific program generator depends on extensive domain analysis, culminating in the formalization of problem domain concepts, corresponding problem specification notations, and a library of solution methods for each domain. All of these activities are formal methods in the broad sense.

The reader is cautioned that not all of the authors use the phrase *formal method* in the broader sense recommended here. For example, Berry states that formal methods do not help in identifying requirements. This is true under the narrower interpretation of the phrase, but not necessarily the broader one.

3. Overview of the Papers

Several concept papers assess the applicability of formal methods to engineering practice. Berry notes that formal methods must be cost effective to be of practical use, that requirements are the central practical issue, and that most formal methods do not help to identify requirements. He also conjectures that formal methods help when they do because they provide a second iteration on conceptual formalization. Robertson analyzes observed failures of formal methods and their causes.

Another group of papers addresses automated reasoning and analysis. Bjorner presents a decision procedure for queues. Manna, Sipma and Uribe describe a method for combining deductive inference and model checking that can provide proofs about infinite state systems using algorithmic finite state methods. Cleaveland and Sims present methods to improve the efficiency of generic, automatically generated model checkers. Narasimha, Cleaveland and Iyer present a model, logic, semantics, and model-checking procedure for probabilistic systems. Kwak, Lee, and Sokolsky give a method for symbolic schedulability analysis that links to efficient equation solvers, which could be used to synthesize designs by solving for values of design parameters that would make the design achieve schedulability guarantees. Berzins analyzes the inference requirements for engineering automation and identifies the need for lightweight inference methods: sound, very efficient, typically restricted or incomplete.

A third group of papers report on engineering aspects and practical experiences in the application of formal methods. Polak reports a successful application of automatic program synthesis in a specialized domain (satellite control systems), and analyzes the reasons for the project's success. Kosiuczenko and Wirsing formalize a common design notation for communication among distributed systems (message sequence charts) using timed rewrite logic, and use the formalism to test a specification by executing it, revealing a fault. Gelfond and Watson describe the application of logic programs with non-monotonic semantics to realize automated decision support for a complex domain (space shuttle operation in the presence of multiple equipment failures). Volker and Kraemer describe the successful application of the higher order logic HOL to the development of a verified library of function blocks for a safety-critical domain (industrial control). Gafni, Feldman and Yehudai present a real-time design language for large scale applications and explain the associated design process via an example (cruise control). Cooke describes a formalism for expressing implicit concurrency in data parallel computation, with applications to data mining. Zhang, Lee, Friedel, and Keyser describe statistical methods for generating facts from raw data to provide decision support for an engineering task (diagnosis and repair of phased array antennas).

Peter Wegner presented the idea that interactive systems fundamentally change the nature of computing, and that this change has far-reaching effects that have not been fully integrated into current theories of computing and engineering science. The main ideas are summarized here because there is no corresponding paper in the proceedings (for more details, see Mathematical Models of Interactive Computing, <http://www.cs.brown.edu/people/pw>). The difference is that the input to an interactive machine is not fixed in advance, and could depend on the partial output produced by the machine up to that point. A difference in expressive power due to this effect is claimed. This view of computation leads to different kinds of formal models, such as co-algebras; and different modes of reasoning, such as co-induction, which are relevant to the analysis of open (extensible) systems of the kind common in the current practice of object oriented design. The proposed change in viewpoint stimulated discussion

as well as some controversy about the details and their philosophical interpretation.

4. Summary of the Discussions

The National Science Foundation is considering the impact of the PTAC report (<http://www.ccic.gov/ac>) and its impact on national research priorities, as summarized below. The report's major recommendation was to make software research an absolute priority. The four major research priorities identified are:

- (1) Software
- (2) Scalable information infrastructure (networking)
- (3) High performance (peta-flops) computing, including software R & D
- (4) Socio-economic and workforce impacts

The report finds that software demand exceeds the nation's capability to produce it, that we must still depend on fragile software, that technologies to build reliable and secure software are inadequate, and that the nation is under-investing in fundamental software research.

The report makes the following recommendations:

- (1) Fund fundamental research in software development methods and component technology;
- (2) Sponsor a national library of software components in subject domains;
- (3) Make software research a substantive component of every major IT research initiative; and
- (4) Fund fundamental research in human/computer interfaces and interactions.

Relevant research initiatives include ASCI (Accelerated Strategic Computing Initiative) and NGI (Next Generation Internet). The internet is making the next step, with major implications for software research. Yesterday's environment is not tomorrow's, and many issues need rethinking within the future context.

We are at a unique point in IT history: agendas are being set and recommendations are being made. The field needs a research agenda, a plan for research management, and action to build public support. Consequences of not acting include negative economic impact and loss of global leadership and competitiveness. One issue is that we are not currently able to meet the demand for software. We therefore need to:

- (1) empower end-users with domain-specific tools that create software;
- (2) make component-based development a reality;
- (3) automate software engineering processes; and
- (4) produce more well-trained professionals.

Another issue is that we cannot produce high-confidence systems, and cannot even produce routine systems routinely. We therefore need to:

- (1) understand what works and what does not;
- (2) understand the science of software construction; and
- (3) create a discipline of software engineering.

The problems identified in the PITAC report have many facets, including unresolved practical problems, rapid change, immaturity of the science, a gap between theory and practice, fragmentation of the research community, and inadequate infrastructure for technology transfer.

The recurring horror story is that we can not afford to build software systems using current technology. This has been true for many years despite improvements in the state of practice. We have not made a convincing case that we have done much. Some of the reasons for this are increasing demand and rapid change, lack of effective technology transfer, and lack of the right kind of science.

The practice of software engineering is moving very fast, in an attempt to keep up with demand and stay ahead of the intense competition. Time to market is vital in the commercial world. Many developers jump on aggressively marketed software fashions, although they often include ad hoc

methods and worst practices along with some improvements.

Despite these difficulties, the commercial world has made progress. For example, Java is an improvement over previous practice. Networking and communication are coming together, and succeeding in reusing resources. Commercial systems engineering is improving. We can successfully educate professionals in about ten years.

Other commercial steps have been less effective. UML had the benefit of lots of talent with inconclusive results. The semantics of C++ remains controversial. Component technology is in fashion although it is still difficult to make components work together.

There is a widespread attitude in the commercial world that academic results are impractical and that theoretical results take too much time and cost to incorporate into practice, especially in a highly competitive world. Some parts of the theoretical computing community take the attitude that practical engineering is irrelevant. The result is ineffective technology transfer and engineering practice with a weak scientific basis.

This is an area where improvement is possible. Instead of a struggle between theory and practice, there should be a supply chain, and a coherent vision of problems flowing up the supply chain and solutions flowing down the supply chain. This should be a continuous, orderly, and effective process. Currently, it is not. We can not afford change in random directions.

There are multiple causes for the current situation, including immaturity of the discipline. The problem goes deeper than a lack of communication that could be resolved by the current practices of our educational systems. Many issues that arise in engineering practice have not been addressed by the scientific community. There is growing awareness of these issues and increasing resolve in the scientific community to address them by developing a more robust and principled basis for future software engineering technologies.

Past emphasis on formal methods in response to this problem has been a mistake. We should instead speak of and insist on effective, rational methods to achieve goals. The Latin for *method* is "via ratio," a rational path. It is not convincing to say, "We are on the right side because math and formulas are what matters." A shift of paradigm is now needed. The quality of the result and the cost of producing that result are what matter. For progress in engineering, it is essential to automate the process. The solution must be a highly interactive, adaptive, automated system. We must admit that, even if we build an advanced system, it will be at a cost of not doing it again.

As science is currently inadequate to support automated engineering, our community needs to understand and develop the science needed to bring the engineering to this level. Formalization is useful to the degree that it contributes to this goal by enabling automation or systematization of engineering processes.

There are two kinds of science: theoretical science focuses on understanding and prediction, while engineering science focuses on empirical validation of theory-based predictions, and learns mostly from failures - as, for example, in seismology. A finer interplay between mathematics and empirical science is needed to achieve progress. Many good ideas have been proposed, but often without a plan to evaluate success. The only basis for rational judgement is empirical science. Many ideas that sound good in the abstract can not be realized in practice. Good empirical computer science is needed, but no one has been able to do it well so far.

To focus effort where it is needed, it may be useful to distinguish engineering science from theoretical science. Recognition of the category *engineering science* is important because research funding agencies typically support science rather than engineering. The aim of engineering science is to improve the capabilities of practicing engineers. The aim of theoretical computing science is to improve our understanding of computing. Automation is a primary goal for engineering science, but not necessarily for theoretical computing science.

Advances in theoretical computing science can contribute in the long term to software engineering by providing better conceptual models and better principles that can be used to build tools for engineers. However, significant effort is required to identify, reformulate, extend, and package the relevant results from theoretical computer science to make them useful for engineering. For example, theoretical advances are often made using simplified models that avoid issues and details that are

inescapable in practical engineering. These issues are in the realm of engineering science, and are vital for progress.

We need technology transfer from relevant new engineering science to make things work. Nobody has the responsibility for this now. There should be an "Expedition Center" to envision what the world is going to be like in 100 years, and a "Transfer Center" to transform those visions into reality. We have to be careful about what kind of technology we transfer: it must be relevant to practical problems. There is much irrelevant material from former type theoretical computer science and others: e.g., How do you get a theorem to find oil?

The various parts of the community must interact more closely than they have in the past to achieve practical impact. Software isolation is a problem. Much software is connected to communication, hardware, and other components. If we do not include these components, we have not solved the problem. Results from other disciplines are relevant also. Software development is a special case of product development. Software is hard because it is abstract; it cannot be visualized. We can learn much from design theory and product management.

Rapid change affects the scientific community as well. The nature of computing may change substantially in the 21-st century. For example, new models of interactive computing and quantum computing are on the horizon. Today's computing environments can not and will not be the environments of tomorrow. Computing is a relatively new science. There is opportunity, but also a need to educate people about what computer science is and what it can be. There is also need for periodic reality checks to ensure feasibility of long-term visions. These exercises can help improve the credibility of our field, can provide course corrections for research agendas and can evaluate readiness for technology transfer as we learn more about what can be done at what cost. DARPA and other agencies have challenge problems that could serve these functions. For example, the automatic theorem proving community has a standard set of benchmark theorems.

There is a tension between long-term goals and short-term goals. Funding agencies require that goals be achieved on a yearly basis. This is an issue that must be faced by all branches of science, not just in computing. We can ask how the issue is handled in other disciplines, such as particle physics. Physics has a history of setting up visionary programs. In Italy, 72 percent of money for basic science goes to physics, and only 1.7 percent goes to information science. Why is this - a good part of the answer is that the physics community behaves in a political way, i.e., it has a lobby. They say, "We have this great vision. We need Congressional funding for astrophysics, etc.", and then set up a lobby and get real money. We need to develop a similar vision and agree to work together toward that vision.

In computing science, we have not agreed on the goals. This has been aggravated by the rapid rate of change, which has spawned computing schools of thought, and intense competition for scarce research support. We need to identify our goals and stick together, instead of "dissecting ourselves to death." Computing research does not have to be a zero sum game. The goals identified in PITAC report are a good starting point for developing a shared agenda for the entire computing community.

Computing is the most successful technical discipline, in that it has come to relevance and has been applied in a relatively short time. Decidability and computability ideas appeared only at the beginning of this century. We had a vision of software engineering in 1968, but people were not aware of how much is hidden behind that vision. The digital point of view brought in a whole new view of the world, as opposed to physics. There is a basic difference between the root of physics and the root of computer science. The foundations of computer science are very simple - i.e., Turing machines suffice, with some modifications. NP completeness is not the most central problem. The real problems come on the macro level, in building systems and with human factors. The roots of physics are different, more involved. The theory of digital models may become much more than it is today.

We should be happy to work in a scientific field that has such a high level impact. We should also understand that there is a real push in progress, and appreciate that scientific push. What we have done wrong is to engage in too much infighting, much of which is due to not understanding the inherent positions imposed by the disciplines of our colleagues. What we have gained over the last 20 years could not have been done without deeper understanding. What we actually do in practice is not called *formal methods*, yet we have made more progress than we realize. It is important to make the field

more transparent. We are just at the beginning.

5. Conclusions

The technical presentations and the engineering experiences reported at the workshop support the premise that engineering automation can lead to significant practical gains. Some of the papers detail the circumstances under which such gains can be realized using currently known techniques, thus providing a snapshot of the current state of the art in the area.

Another outcome of the workshop was a change in the attitude of the participants. For the first time there appeared a broad consensus that we should work together and agree on a larger common vision that we can all contribute to from our individual specialties. Most participants accepted the idea that theoretical work should contribute to engineering over a medium- to long-term time horizon. A working approximation to that vision is the improvement and application of computing science to, in turn, improve and automate processes for developing reliable computer-based systems.

This consensus suggests a direction for action. The common vision needs to be supported by a more detailed research and development plan, providing explicit intermediate goals on the way toward the ultimate end. We should interleave our specialized scientific efforts with periodic application and integration of results from our different disciplines, with assessment steps and identification of unsolved problems that lie between the solved fragments, and with validation and adjustment of the assumptions used as the basis for the next round of basic research. Applications of new and sometimes deep theories rarely happen spontaneously. For best success, those researchers who originate new theories should spend part of their effort identifying and developing applications of those theories, perhaps in cooperation with groups whose primary focus is empirical engineering science. Some of our most valuable lessons have come from the analysis of failed attempts to apply existing theories.

We must work together to agree on how these threads will fit together into a coherent whole, and to form a more detailed vision that addresses society's long-term needs. Technology transfer and public relations are part of this puzzle. We need to better communicate to the public how engineering automation and the basic research it will take to achieve that goal will alleviate the difficulties associated with computer-based systems that currently touch all of our lives. We need to make concrete progress in this direction, and to demonstrate the practical impact of that progress in a systematic and coordinated way. It is important to put past disagreements behind us to work together for the common good of both the computing discipline and society at large.

Formal Methods: The Very Idea*, Some Thoughts About Why They Work When They Work

Daniel M. Berry§

Computer Science Department, University of Waterloo
200 University Ave. W., Waterloo, Ontario N2L 3G1, Canada
FAX: +1-519-746-5422, E-mail: dberry@csg.uwaterloo.ca

ABSTRACT

The paper defines formal methods (FMs) and describes economic issues involved in their application. From these considerations and the concepts implicit in "No Silver Bullet", it becomes clear that FMs are best applied during requirements engineering. A explanation of why FMs work when they work is offered and it is suggested that FMs help the most when the applier is most ignorant about the problem domain.

Keywords: Formal Methods, Verification, Refutation, Economics, Requirements Engineering, Second-Time Phenomenon, Ignorance Hiding, Evangelists

1 INTRODUCTION

This paper is something that I have been meaning to write for some time now. I have been giving a talk whose title is that of this paper to a variety of audiences. In each case, the discussion generated was interesting and supplied more material for the ever growing talk. When I received the invitation to attend the Monterey Workshop on Engineering Automation for Computer-Based Systems, I saw an opportunity to present the talk to an audience of almost entirely formal methods people, including some of the pioneers. The talk was much better received than I thought it might be given its controversial nature. Moreover, all speaking participants at the workshop were required to produce a paper for the proceedings. The paper that I wanted to write is the result.

Because the paper represents more my personal opinion rather than some rigorously established scientific conclusion, the paper uses first person when referring to the author.

I have benefited particularly from an electronic discussion in 1995 with Martin Feather.

1.1 Author's Background

My current area of research is Requirements Engineering (RE) [5]. The focus of this area is on how to get requirements for software-intensive computer-based systems (SWICBSs). It is now recognized, as is explained in the body of this paper, that determining the requirements for SWICBSs is the hardest part of their development, and difficulties in determining these requirements are the source of a vast majority of errors found in delivered SWICBSs. The RE area is interested in understanding *why* a method of determining requirements works when it does and *why* a method of determining requirements fails when it does.

1.2 Outline of Paper

This paper starts of with a brief definition of FMs. It then gives some feeling for the economics of applying FMs to the development of SWICBSs. Fred Brooks's observation of "No Silver Bullet" is recalled for what it says about the difficulty of determining SWICBS requirements. The paper offers that the most useful time to apply FMs is in the requirements engineering phase of SWICBS development. Not all applications of FMs lead to high quality SWICBSs. However, when they do succeed, there appear to be two factors working for that success, the second time phenomenon and qualities of the people who push for and engage in FMs.

*with apologies to James H. Fetzner [14]

§ on sabbatical leave from Computer Science Department, Technion, Haifa 32000, ISRAEL, with part of work done at GMD FIRST, Rudower Chaussee 5, 12489 Berlin, Germany

So as not to lose readers who believe in FMs and who see parts of this paper as arguing against their use, please consider that I believe in FMs and use them when appropriate. I used to work for a company that sells FM technology and applies FM to clients' SWICBS development problems, including for secure operating systems. Moreover, I did some fundamental work on the underlying theory of one FM a long time ago [3]. The reader will see that I am generally in favor of FMs, but there are serious problems of which we must be aware. The paper offers some unconventional ideas as to why FMs are successful when they are.

2 DEFINITIONS OF FMs

Basically, an FM is any attempt to use mathematics in the development of a SWICBS, in order to improve the quality of the resulting SWICBS. For the purpose of this paper, I am trying to include in the realm of FMs anything anyone working in FM claims is an FM. If I have neglected to include one, my apologies. Please inform me by e-mail and include a reference to literature about it. For fuller discussions, see the papers by Hall, Leveson, and Wing [18, 22, 26], and papers cited therein.

There are three main groups of FMs, verification, intensive mathematical study of key problem, and refutation. Each group is described; its strengths, weaknesses and costs are considered,

2.1 Verification

The first group of FMs are those that attempt to provide a basis by which the software of a SWICBS can be formally proved to be a correct realization of a specification embodying its requirements. Strictly speaking, the code is proved consistent with a formal specification. Rarely, however, is the full proof carried out. Nevertheless, the FMs in this group all have as their goal the production of at least one part of a full proof of correctness. Within this group, there are many levels of formality and completeness. Here, by "completeness" is meant that of application and not that of a mathematical theory. Some of the FMs of this group can be characterized as some collection of levels of Table 1. In this table, the notation " $P \leftrightarrow C$ " means "partial through complete", "C" means "complete", and "P" means "partial".

1	$P \leftrightarrow C$	formal specification of requirements
2	$P \leftrightarrow C$	verification of consistency and basic correctness of requirements specification
3	$P \leftrightarrow C$	formal specification of design
4	$P \leftrightarrow C$	verification of consistency of requirements and design specifications
5	$P \leftrightarrow C$	formal specification of code
6	$P \leftrightarrow C$	verification of consistency of (requirements), design, and code specifications
7	C	code
8	$P \leftrightarrow C$	verification of consistency of (requirements, design, and) code specifications and the code

Table 1

In each verification level, the items in parentheses are included in what is verified to be consistent by virtue of transitivity provided by lower level proofs. Only the items outside the parentheses are directly involved in that level's proof. In Level 2, "basic correctness" means verification that the requirements specification satisfies any available independent specification of the correctness criterion, e.g., security.

A typical FM in the group described by Table 1 consists of some collection of levels. Sometimes only Level 1 and Level 7 are carried out with no verification. Rather, the doing of the formal specification allows much to be learned about the SWICBS to be developed before carrying out the actual development. Much more is learned this way than when only an informal, natural language specification is written. Sometimes only Level 1 is carried out for the purpose of documenting the requirements of the SWICBS, on the grounds that a formal specification is the most precise.

Applying FMs of this group drives up the cost of SWICBS development as high as 2 fold over applying normal systematic methods of writing just the code if only Levels 1 and 7 are carried out, 10 fold if Levels 1 through 4 and then 7 are carried out, and even higher if more verification is carried out. These are the rules of thumb that were applied in the company mentioned above that sold FM technology.

2.2 Intensive Study of Key Problems

The second group of FMs are the intensive mathematical study of one or more difficult aspects of the SWICBS. These are an attempt to avoid the heavy costs of the verification FMs. Rather than specifying the entire SWICBS, only the difficult or problematic parts of the SWICBS are considered. For example, if the job is to build a secure operating system that guarantees that only authorized users gain access to any specific file, instead of specifying the whole operating system and proving its security, one could focus on the security-relevant portion of the system at the specification, design, or code level, or at some combination of these. While this focusing is considerably cheaper than dealing with the full system, it is fraught with a serious danger of overlooking something that is security relevant. One cannot be certain that the ignored portions of the system are not security relevant, that they do not impact and they are not impacted the parts designated as security relevant and therefore under study. To prove that the ignored parts are not security relevant turns the FM into a costly verification. Nevertheless, as one gets experience with a class of applications, he or she becomes more certain about what can safely be ignored.

- | | | |
|---|---|--|
| 1 | C | study of one difficult aspect of requirements e.g., security, safety |
| 2 | C | study of one difficult aspect of design |
| 3 | C | study of one difficult aspect of code |

Table 2

I would classify into this group any development in which theoretical knowledge is used to make the development of a program more systematic. The most common example of this phenomenon is compiler writing, which borrows heavily on the theory of phrase-structure grammars and has spawned its own collection of theory.

The rules of thumb that I have heard are that these intensive study FMs drive the development cost up from 2 through 5 fold, depending on the complexity of the problem and depending on how many levels of the study are carried out.

2.3 Refutation

The elements of the third group of FMs take an entirely different approach, that of refutation rather than verification [24], that is, instead of trying to prove that the SWICBS meets its requirements, one tries to refute the claim that it does. The advantage is that the correctness claim can be refuted by one counter example, while a proof must consider all possible cases. There are two kinds of refutation. One kind are those based on computable properties of some specification of the SWICBS. The second kind are those based on execution of some specification of the SWICBS on test data. Note that neither of these can be verification of correctness; correctness is not a computable property, and testing can be used to show the presence of errors, never their absence [11].

Given a specification of the complete SWICBS, as in Level 1 of the verification group of FMs, two examples of computable properties that can be used to refute the claim of correctness are:

- type checking in the specification
- cross reference checking in the specification

A type error, undefined identifier, or defined, but unused identifier can be the symptom of a more serious error, which a thinking human being should be able to spot given the evidence.

Given a specification of the complete SWICBS, as in Level 1 of the verification group of FMs, if the language of the specification is executable, e.g. OBJ [16] or INATEST [20, 12] one should be able to execute the specification either with actual data or symbolically. The execution with test data may show errors in the specified SWICBS. Alternatively, one might be able to build a finite state model of the SWICBS, either directly or by simplifying the model or abstracting away some of the complexity. Execution of the finite state model with test data can show errors. In addition, if the model is finite state, there are computable checks, e.g., reachability analysis, that can be used to show the presence of problems. This group of activities is called model checking.

While locating an error by refutation is not guaranteed, in practice, model checking does expose errors, just as does execution of the finished SWICBS. However, as is shown in Section 5, it is highly preferable for an error to be exposed early in the life cycle than later after deployment of the finished SWICBS.

The refutation approaches cost that of Levels 1 and 7 of the verification group plus only 5-50% for the refutation. That is, refutation drives the cost of development up to between 2.05 and 2.50 fold, and this is cheaper than with full verification.

2.4 Programming Itself as an FM

Remember that a program itself is a formal specification. A programming language is a formally defined language with precise semantics just like Z, in fact, even more so than Z, which purposely leaves some things undefined. One could not prove the consistency of specifications and code if code were not formal. Therefore, programming itself is an FM in the sense that writing a formal specification is an FM. Remember that programming is building a theory from the programming language and library of abstractions, i.e., the ground, up, just like making new mathematics.

2.5 General Limitation of FMs

An ultimate problem with any of these FMs that are based on a specification of the SWICBS under design is the accuracy of this specification. If it does not specify what is desired, that is, it is not *right*, code that is consistent with this specification does not do what is desired. The specification could be wrong for an error of commission or an error of omission. The specification could deal with a given situation in an inappropriate way, e.g., shutting down an aircraft engine that is in an inconsistent state. The specification could fail to deal with an issue entirely or could even fail to detect the presence of the issue.

2.6 Economic Realities of FMs

For most software, it is just not worth the cost to apply FMs; one can get more than acceptable quality by inspection [13] for up to 15% more and absolutely superb results by just doing the software twice at the cost of about 100% more. However, for highly safety- and security-critical systems, for which the cost of failure is death or is considered very high, FMs are necessary to achieve the required correctness and are well worth the cost.

David Notkin [9] observes about model checking that sometimes it is necessary to make simplifying assumptions in the model to get a model tractable enough to be checked. This necessity creates a dilemma. Without simplification, the specification cannot be analyzed and critical problems might be overlooked. However, simplifications might hide critical problems, especially as abstraction is used to collapse a number of states into one. In the end, it is an issue of costs. Which problems cost more, the ones overlooked by lack of analysis or the ones overlooked by simplification?

On the other hand, there is evidence that careful use of FMs during RE of a SWICBS can eliminate enough errors from ever showing up later in the development of the SWICBS, when they are very expensive to fix, that the cost of the later development is reduced enough that the total cost of an FM-assisted development is no more or even less than that of an unassisted development [18, 19]. See Section 5 for more information about the cost to fix errors as a function of the development stage in which they are found.

3 MOST ERRORS INTRODUCED DURING REQUIREMENTS SPECIFICATION

One thing that has been learned over the years is that most errors are introduced into SWICBSs during the requirements discovery, specification, and documentation stages, to the tune of between 65 and 85%. The coding stage introduces only about 25% of the errors ever introduced into a SWICBS [6]. Verification of the consistency of code to specification is by far the most expensive FM. Therefore, it is not clear how useful code verification is if only 25% of the errors are introduced during coding, and these errors are probably the easiest to fix. It seems that it is more cost effective to spend just 15% more for inspections than to spend more than 10 fold to fix errors introduced during coding. Therefore, the focus of FMs must be on requirements.

4 NO SILVER BULLET (NSB)

Not so long ago, Fred Brooks observed that, with respect to software development, "There's no silver bullet" [8] that will suddenly and miraculously make programming fundamentally easier than it has been. He classifies software difficulties into two groups, the *essence* and the *accidents*. The essence of building software is devising the conceptual construct itself. This is very hard, because that conceptual construct is of arbitrary complexity, it must conform to the given real world, it constantly changes, and it is ultimately invisible. On the other hand, most productivity gains have come from fixing accidents. The accidents are the current technology that is used to develop the software. Examples of such accidents and their solutions are:

- really awkward assembly language eliminated by high-level languages,
- severe time and space constraints eliminated by the introduction of big and fast computers,
- long batch turnaround time eliminated by time-sharing operating systems and personal computers,

- tedious clerical tasks for which tools are helpful eliminated by those tools, such as make, rcs, xref, spell, grep, fmt, and
- the drudgery of programming user interfaces eliminated by tools for building graphic interfaces such as X Windows, Java, Visual Basic and other GUI libraries.

These have been significant advances, and they have made coding significantly easier and less error prone. However, again, these advances attack only the minority of errors introduced by coding and do nothing about the essence.

Unfortunately, the essence has resisted attack. We have the same sense of being overwhelmed by the immensity of the problem and the seemingly endless details to take care of, and we produce the same kind of brain-damaged SWICBSs that makes the same stupid kind of mistakes, as we did 30 years ago! The source of these errors is that we just did not understand the conceptual construct that was to be constructed. We overlooked details or have some details wrong.

5 FMs AND THE ESSENCE OF SOFTWARE

Another way to describe the essence is "requirements", not specifications, which are just a statement of requirements, but the requirements themselves. FMs just do not help identify requirements. They do not help us crack the essence.

There is a myth going around. Some FM evangelists claim, "If *only* you had written a formal specification of the system, you wouldn't be having these problems. Mathematical precision in the derivation of software eliminates imprecision." Yes, formal specifications are extremely useful in identifying inconsistencies in requirements specifications, especially if one carries out some minimal proofs of consistency and constraint or invariant preservation. Interestingly, writing a program implementing the specification also helps identify inconsistencies in the specification; programming is another FM.

Contrary to the claim of these evangelists, FMs do not find all gaps in understanding. As Gordon and Bieman observe, omissions of functions are difficult to recognize in formal specifications [17], just as they are in programs. von Neumann and Morgenstern [25] say, "There's no point to using exact methods where there's no clarity in the concepts and issues to which they are to be applied."

Indeed, Oded Sudarsky, in a private discussion over coffee, pointed out the phenomenon of *preservation of difficulty*. Specifically, difficulties caused by lack of understanding of the real world situation are not eliminated by use of FMs; instead the misunderstanding gets formalized into the specifications, and may even be harder to recognize simply because formal definitions are harder to read by the clients. Sudarsky adds that formal specification methods just shift the lack of understanding from the implementation phase to the specification phase. The air-bubble-under-wallpaper metaphor applies here; you press on the bubble in one place, and it pops up somewhere else.

FMs *do* have one positive effect, and it is a big one. Use of them increases the correctness of the specifications. Therefore, you find more errors of commission at specification time than without them, saving considerable money for each bug found earlier rather than later. Remember, the cost to repair an error goes up dramatically as the project moves towards completion and beyond. Figure 1 shows a graph relating the relative cost to repair an error as a function of the SWICBS development stage in which the error is found. Note that the cost scale on the y-axis is logarithmic, and the graph itself looks exponential even on a logarithmic scale. It saves lots of money to find errors earlier, and FMs help find errors earlier. However, these errors are of commission rather than of omission.

Another reason FMs do not help identify requirements very well is that requirements always change—it is inherent in the software—and formalization requires freezing the requirements long enough to write the specification and carry out the verifications. Meir Lehman identifies concept of E-type system [21]. It is a system that solves a problem or implements an application in some real world domain. Once installed, an E-type system becomes inextricably part of the application domain, so that it ends up altering its own requirements. As an example, consider a bank that exercises an option to automate its process and then discovers that it can handle more customers. It advertises and gets new customers, easily handled by the new system but beyond the capacity of the manual way. The bank cannot back out of automation. The requirements of the system have changed from being just optional to being required. Also, daily use of a system causes an irresistible ambition to improve it as users begin to suggest improvements. Who is not familiar with this phenomenon, either as a customer or as a developer? In fact, data show that most maintenance is not corrective, but for dealing with E-type pressures! See Figure 2. Formalization of the requirements does nothing to make the details of these kinds of changes more predictable.

6 SECOND TIME PHENOMENON

In 1985, I published a paper with Jeannette Wing that suggests that FMs work, not because of any inherent property of FMs as opposed to just plain programming, which is really also an FM, but rather, because of the second time phenomenon [2]. If you do anything a second time around you do better, because you have learned from your mistakes the first time around. Indeed, Fred Brooks says: "Plan to throw one [the first one] away; you *will* anyway!" [7] In other words, you cannot get it right until the second time. If you write a formal specification and then you write code, you've done the problem

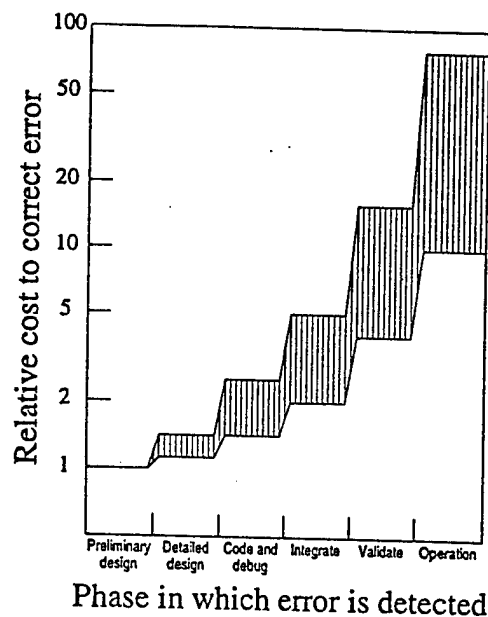


Figure 1

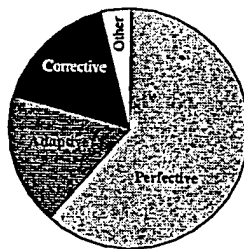


Figure 2

formally two times. Of course, the code will be better than if you had not done the formal specification. It is the second time! Note that writing an informal specification and then writing code does not have the same effect. It is too easy to handwave and overlook details and thus fail to find the mistakes from which you learn. It has to be two formal developments, specifications or code, for the second-time phenomenon to work.

Observe how the two-time approach is requirements centered. One is not going to fix implementation errors this way, because the second time is not the same implementation as the first time. Even if they were the same, one can introduce new errors in the rewrite. The focus of the first specification or coding effort is on understanding the essence and eliminating requirements errors. The focus of the second is on implementing the understood essence. As Euripedes says, "Second thoughts are always wiser".

7 THE IMPORTANCE OF IGNORANCE

In a recent article, "Importance of Ignorance in Requirements Engineering" [4], I report on my and Orna Berry's experiences in practicing ignorance hiding [1] in requirements engineering. I observed that I seem to do best when I am in fact most ignorant of the problem domain. For example, I had been called in as a consultant to help a start-up write requirements for a new multi-port Ethernet switching hub. I protested that I knew nothing about networking and Ethernet beyond nearly daily use of telnet, ftp, and netfind. At one point, earlier in my life, I had worried that the ether in Ethernet cables might evaporate! Despite my ignorance, I did a superb job, in fact, better than I normally do in my areas of expertise. By being ignorant of the application area, I was able to avoid falling into the tacit assumption tarpit. The experience seems to confirm the importance of the ignorance that ignorance hiding is so good at hiding. It was clear to me that the main problems

preventing the engineers at the start-up from coming together to write a requirements document were that all were using the same vocabulary in slightly different ways, none was aware of any other's tacit assumptions, and each was wallowing deep in his own pit. My lack of assumptions forced me to ferret out these assumptions and to regard the ever so slight differences in the uses of some terms as inconsistencies.

My conclusion is that every requirements engineering team requires a person who is ignorant in the application domain, the *ignoramus* of the team, who is not afraid to ask questions that show his or her ignorance, and who will ask questions about anything that is not entirely clear. It is not claimed that expertise is not needed. On the contrary, experts are needed to provide the material in which to find inconsistencies. Also, there is a difference between ignorance and stupidity; the *ignoramus* cannot be stupid. On the contrary, he or she must be an expert in general software system structures and how computer-based systems are built. He or she must be smart enough to catch inconsistencies in statements made by experts in fields other than his or her own, inconsistencies in their tacit assumptions, to abstract well, and to get to the bottom of things. Most importantly, he or she must be unafraid to ask so-called stupid questions to expose all tacit assumptions. (This is part of smartness since usually stupid people are afraid to ask stupid questions for fear of exposing their stupidity.)

The final recommendation is that each requirements engineering team needs at least one domain expert, usually supplied by the customer and at least one smart *ignoramus*.

As a consequence of these observations, resumes of future software engineers will have a section proudly listing all areas of ignorance. This is the only section of the resume that shrinks over time. The software engineer will charge fees according to the degree of ignorance: the more ignorance, the higher the fee!

Soon after publication of the Importance of Ignorance paper, I received an e-mail letter from Martin Feather. He wrote,

I have often wondered about the success stories of applications of formal methods. Should these successes be attributed to the formal methods themselves, or rather to the intelligence and capabilities of the proponents of those methods? Typically, proponents of any not-yet-popularised approach must be skilled practitioners and evangelists to [help bring the approach] ... to our attention. Formal methods proponents seem to have the additional characteristic of being particularly adept at getting to the heart of any problem, abstracting from extraneous details, carefully organizing their whole approach to problem solving, etc. Surely, the involvement of such people would be beneficial to almost any project, whether or not they applied "formal methods."

Daniel Berry's contribution to the February 1995 Controversy Corner, "The Importance of Ignorance in Requirements Engineering," provides further explanation as to why this might be so. In that column, Berry expounded upon the beneficial effects of involving a "smart *ignoramus*" in the process of requirements engineering. Berry argued that the "*ignoramus*" aspect (ignorance of the problem domain) was advantageous because it tended to lead to the elicitation of tacit assumptions. He also recommended that "smart" comprise (at least) "information hiding, and strong typing ... attuned to spotting inconsistencies ... a good memory ... a good sense of language..." so as to be able to effectively conduct the requirements process.

Formal methods people are usually mathematically inclined. They have, presumably, spent a good deal of time studying mathematics. This ensures they meet both of Berry's criteria. Mastery of a non-trivial amount of mathematics ensures their capacity and willingness to deal with abstractions, reason in a rigorous manner, etc., in other words to meet many of the characteristics of Berry's "smartness" criterium. Further, during the time they spent studying mathematics, they were avoiding learning about non-mathematics problem domains, hence they are likely to also belong in Berry's "*ignoramus*" category. Thus a background in formal methods serves as a strong filter, letting through only those who would be an asset to requirements engineering.

8 THE LAST LAUGH OF NATURE

Don Gause [15] points out that there are two kinds of people involved in the development of any SWICBS, developers and customers. Each person divides the universe of discourse (UoD), the domain of the SWICBS, into two parts, what he or she knows (K) and what he or she does not know (DK). The effect of these orthogonal partitions of knowledge divides the UoD into four parts as is shown in Figure 3. The problem is that we do not know the size of the DKs. We like to think that after studying the problem a long time and, the DKs have been reduced to a tiny fraction of the Ks. However, the DKs could be bigger than the Ks like the proverbial tip of the iceberg. Even if, in fact, the DKs are small compared to the Ks, the DKs can never be eliminated. The parts of the DKs that cannot be eliminated are called "nature's last laugh" by Don Gause. Examples of nature's last laugh include cold fusion and all previously accepted but now discredited theories of the universe.

		Developers	
		Know	Don't Know
Customer & Users	Know		
	Don't Know		Nature's Last Laugh (DK×DK)

Figure 3

The importance of the ignoramus comes through loud and clear. Every RE team requires a smart ignoramus relative to the real world domain of the system under design, who is not afraid to ask questions to reduce his or her DK in an attempt to get his or her K to include the client's and users' K. He or she must not be stupid; in fact, he or she must know enough about system architecture to be able to formulate enough of a model to prompt the questions.

Maybe this is the role of FMs, to increase the Ks, but that is all it can be. It must be accepted that there will always be the DK, nature's last laugh that no one will find unless someone is lucky enough to ask the right dumb question.

9 ANOTHER EXPERIENCE

A complementary paper in these proceedings by David Robertson [23], considers how attempts to use FMs in the early stages of SWICBS design can fail. He describes his experiences trying to teach applied mathematicians to apply temporal logic to specifying a reactive system. The experience is more evidence of the importance of ignorance in writing specifications.

Robertson's group was collaborating for the first time with a group of computer-using applied mathematicians that knew temporal logic theory inside and out, but had never applied this theory to specify any system. They were asked to specify in temporal logic a domain that practically invited temporal specification. As Robertson described it in e-mail to me, it "was an obvious application which could be done in a short time using simple temporal relations and forms of inference which were pedestrian by comparison to those with which the temporal logic group is familiar." After an initial failure to specify, the mathematicians asked Robertson, with some embarrassment, to write a prototype for them. He rapidly turned out a Prolog program, of less than 100 lines of the kind that a bright second year undergraduate should be able to write. This prototype proved to be enough of a trigger, and the mathematicians are now happily turning out specifications of more complex systems.

The mathematicians simply could not take the first step without something concrete to help them. Robertson believes that the difficulty was that they lacked training in problem representation. As happens with students who are unable to apply the theory they learn to problems, the mathematicians had not developed any intuition about how to abstract away the details of a complicated problem in order to get a useful specification. Robertson believes that "it is often easier to produce the sort of idealised system I described above if you are 'just ignorant enough' about logic not to be drawn into too complex modelling at an early stage but 'just smart enough' not to make logical goofs and to be able to transfer the initial prototype to more experienced hands. People in this line of work need to retain a certain amount of wide-eyed ignorance of the domain — otherwise they would be tempted to model problems too deeply, which is fun but seldom profitable."

10 SOCIAL PROCESSES AND FMs

It is also my belief that the proper context for FMs in the development of SWICBS is in the highly social process of requirements engineering. This recalls the 1979 DeMillo, Lipton, and Perlis paper, "Social Processes and Proofs of Theorems and Programs" [10]. They observed that mathematical proofs work because of the social processes in and around them that help to insure that only correct theorems get published (and even then they are not all correct). They argued that the proofs required by FMs applied to programming are generally carried out by grunt mathematicians working alone and without the benefit of social interaction, because, unlike publishable proofs in mathematics, proofs about programs are quite simply and frankly boring. Bored grunt mathematicians make mistakes. Proofs without social processes are not trustworthy enough for the needs of systems critical enough to justify the cost of FMs.

11 CONCLUSIONS

It is my belief that FMs work when they work, not so much because of formality, but rather because of

1. what is learned when applying FMs, that can be applied in the next round of development and

2. the nature of the people who willingly and enthusiastically apply FMs.

Despite the weakness of FMs at discovering new requirements, FMs work best when they are being applied to the RE stage of SWICBS development to help understand and correct the requirements.

ACKNOWLEDGMENTS

I thank Jo Atlee, Don Cowan, Steve Easterbrook, Martin Feather, Sol Greenspan, Anthony Hall, Axel van Lamsweerde, Maria Viera Nelson, Dave Robertson, John Rushby, and Meyer Tanuan for comments on an earlier draft.

REFERENCES

- [1] Berry, D.M. and Berry, O., "The Programmer-Client Interaction in Arriving at Program Specifications: Guidelines and Linguistic Requirements," in *Proceedings of IFIP TC2 Working Conference on System Description Methodologies*, ed. E. Knuth, Kecskemet, Hungary (May, 1983).
- [2] {Berry, D.M., Wing, J.M.}, "Specification and Prototyping: Some Thoughts on Why They Are Successful," pp. 117-128 in *Proceedings of TAPSOFT Conference*, Springer, Berlin (March 1985).
- [3] Berry, D.M., "Towards a Formal Basis for the Formal Development Method and the Ina Jo Specification Language," *IEEE Transactions on Software Engineering* SE-13(2), p.184-201 (1987).
- [4] Berry, D.M., "The Importance of Ignorance in Requirements Engineering," *Journal of Systems and Software* 28(2), p.179-184 (February, 1995).
- [5] Berry, D.M. and Lawrence, B., "Requirements Engineering," *IEEE Software* 15(2) (March, 1998).
- [6] Boehm, B.W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ (1981).
- [7] Brooks, F.P. Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley, Reading, MA (1975).
- [8] Brooks, F.P. Jr., "No Silver Bullet," *Computer* 20(4), p.10-19 (April, 1987).
- [9] Chan, W., Anderson, R.J., Beame, P., Burns, S., Modugno, F., Notkin, D., and Reese, J.D., "Model Checking Large Software Specifications," *IEEE Transactions on Software Engineering* SE-24(7), p.498-520 (July, 1998).
- [10] DeMillo, R.A., Lipton, R.J., and Perlis, A., "Social Processes and Proofs of Theorems and Programs," *Communications of the ACM* 22(5), p.271-280 (1979).
- [11] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ (1976).
- [12] Eckmann, S. and Kemmerer, R.A., "INATEST: an Interactive Environment for Testing Formal Specifications," *Software Engineering Notes* 10(4) (August, 1985), *Proceedings of the Third Workshop on Formal Verification*, Pajaro Dunes, CA, February, 1985.
- [13] Fagan, M.E., "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal* 15(3), p.182-211 (1976).
- [14] Fetzer, J.H., "Program Verification: The Very Idea," *Communications of the ACM* 31(9) (September, 1988).
- [15] Gause, D., "Understanding Requirements," Tutorial, Colorado Springs, CO (April 1998).
- [16] Goguen, J.A. and Tardo, J., "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications," in *Proceedings Conference on Specifications of Reliable Software*, Boston (1979).
- [17] Gordon, S. and Bieman, J., "Rapid Prototyping and Software Quality: Lessons from Industry," pp. 19-29 in *Proceedings of the Pacific Northwest Software Quality Conference*, Portland, Oregon (October 7-8, 1991).

- [18] Hall, A., "Seven Myths of Formal Methods," *IEEE Software* 7(5), p.104-103 (1990).
- [19] Hall, A., "Using Formal Methods to Develop an ATC Information System," *IEEE Software* 13(2) (March, 1996).
- [20] Kemmerer, R.A., "Testing Formal Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering* SE-11(1) (January, 1985).
- [21] Lehman, M.M., "Programs, Life Cycles, and Laws of Software Evolution," *Proceedings of the IEEE* 68(9), p.1060-1076 (September, 1980).
- [22] Leveson, N.G., "Guest Editor's Introduction: Formal Methods in Software Engineering," *IEEE Transactions on Software Engineering* SE-16(9) (September, 1990).
- [23] Robertson, D., "Pitfalls of Formality in Early System Design," in *Proceedings of the Monterey Workshop on Engineering Automation for Computer Based Systems*, Carmel, CA (October, 1998).
- [24] Rushby, J., "Calculating with Requirements," pp. 144-146 in *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, IEEE Computer Society, Annapolis, MD (January, 1997).
- [25] von Neumann, J. and Morgenstern, O., *Theory of Games and Economic Behavior*, Princeton University Press, Princeton, NJ (1944).
- [26] Wing, J.M., "A Specifier's Introduction to Formal Methods," *IEEE Computer* 23(9) (September, 1990).

Lightweight Inference for Automation Efficiency*

V. Berzins

Computer Science Department, Naval Postgraduate School
Monterey, CA 93943, USA

Abstract

We explore the role of lightweight inference techniques in creating highly automated engineering support environments for the development of computer-based systems. Lightweight inference techniques are scalable methods for automated reasoning. We outline the types of automation that would be enabled by effective lightweight inference capabilities and survey some promising approaches to realizing the needed capabilities.

1 Introduction

We need improved capabilities for constructing computer-based systems, particularly regarding reliability, cost, development delay, and responsiveness to change. These needs can be addressed by automating some of the design and development work currently done by engineers. This includes analysis, synthesis, and transformation tasks that require reasoning support [12, 8]. This paper explores the types of inference needed in this context, and identifies some key issues for progress.

According to another paper in this proceedings [2], use of formal methods costs 2-10 times more than just producing code. That analysis assumes conventional processes where software systems are developed one at a time by economically separate projects. In that context, the analysis suggests that formal methods are economically justified only for products where the cost of software failure is very high. This picture is not very promising for cost-effective production of high quality software.

An alternative path to cost-effective quality software is to amortize formal methods effort over development of many systems. For greatest benefit of this strategy, we need reliable generators that can produce reliable software for many related applications. This reuses parts of the formal methods effort spent on the critical requirements determination aspect [20] as well as on conceptual modeling, software architecture, and verification of the program generation patterns to realize the architecture. The systems created by a generator can be different products in the same application domain, or they can be improved releases of the same product. The latter pattern is economically significant because the bulk of software cost is attributable to software evolution rather than to development of new systems.

A key benefit of the reliable generator approach is systematic and non-decreasing improvement in software quality, both in the requirements aspect and in the correctness of code with respect to requirements. The approach addresses requirements by reusing domain knowledge and corrective feedback from prior applications [11]. It reduces the particularly problematic errors of omission because requirements issues identified in previous applications of the domain can be systematically checked. It enables monotonic improvement of program reliability, because once a bug in a program generation pattern is fixed, it stays fixed for all future applications of the pattern. As a limiting case, if the patterns can be proven correct, then all future applications generated by the patterns will also be correct, without need for any further proofs unless new rules are added. Automatic generation of the application programs is necessary for success, to prevent human error in the application of the certified program generation rules. Automatic tools at the requirements level are needed for the same reason.

Lightweight inference addresses issues on the critical path to this vision. Automatic inference is needed to realize many parts of the automatic tools. For example, inference is needed to check the applicability

*This research was supported in part by U. S. Army Research Office under contract/grant number 38690 and in part by the the Army Research Lab under grant number 7DNAVYR010.

conditions associated with each program generation rule, to determine which rules are applicable to a particular problem instance, and to decide which is most beneficial if more than one generation rule applies. Inference capabilities are also needed in engineering automation for synthesizing, transforming, and checking the program generation rules, architectures, and requirements models.

Inference has been studied for many years in the context of philosophy, logic, and mathematics. This work has addressed many theoretical issues such as soundness, completeness, and decidability of various inference systems. These results have contributed a great deal to our general understanding of logic and inference.

Some of these general issues, such as soundness, are relevant to our goals. An inference system is *sound* if only valid statements can be proved.¹ Soundness is essential for engineering inference. Automated design processes must give dependable results before engineers will stake their reputations on them.

However, other issues emphasized in mathematical logic differ from those most relevant to engineering automation.

1.1 Inference in Mathematics

Formal systems for inference are part of the foundations of mathematics, and have been studied extensively in the context of mathematical logic. Logicians are interested in deductive power, and have established widely accepted criteria such as completeness and decidability.

An inference system is *complete* if every valid statement has a proof. In logic, completeness of an inference system is a commonly accepted indication of optimal deductive power: it says every true statement has a proof. In the abstract completeness is an attractive goal. Note that completeness is concerned with existence of a proof, rather than whether there is a way to check if a proof exists or how long the proof might be.

A logic is *decidable* if there exists a procedure that will determine whether any well formed statement of the logic is true or false in a finite number of steps. Any question that can be formulated in a decidable logic can in theory be answered by an automated process. This criterion is also an idealization, because it accepts any procedure that is guaranteed to terminate for all inputs, regardless of how many steps it may take. Decidability would be a practical criterion of deductive power in a world where clients have infinite lifetimes.

1.2 Inference for Engineering Automation

Inference supporting engineering automation for computer based systems must face practical concerns.

Most logics useful for software modeling are not decidable. For example, first order predicate calculus becomes undecidable if augmented with standard interpretations for data types that commonly appear in software, such as integers or lists. It is therefore difficult to address our subject matter within the confines of known decidable systems.

Since engineering applications demand soundness and inference systems that are both sound and complete do not exist for undecidable logics,² incomplete inference systems are highly relevant for engineering automation.

Even for engineering problems that can be expressed in a decidable logic, we must face the issue that decision procedures typically have at least exponential running times, even for the simplest and weakest logics. Software analysis and synthesis problems encountered in practice are large, typically millions rather than tens of lines of code. Practical efficiency constraints rule out exponential algorithms at this scale, unless we can partition large practical problems into independent parts with (small) constant bounds on the size of the largest indivisible subproblem.

Most software analysis problems are not decidable if we insist on perfect solutions for all possible programs. Fortunately, we only have to solve the problems that occur in engineering practice, not *all* problems expressible in common logic or programming languages. There has been little success in inventing languages

¹Valid means true for all possible models, i.e. for all possible interpretations of the symbols in the statement that do not have predefined meanings in the logic.

²A decision procedure for the closed sentences of a two-valued logic with a sound and complete inference system can be obtained by enumerating all theorems until either the closed sentence or its negation appears.

that can express the problems arising in software engineering practice but do not have the additional capability to express many intractable problems that do not arise in practice. This puts a premium on efficiently solvable special cases and safe approximations that cover the situations arising in engineering practice. For example, compilers are commonly designed to issue error messages for all cases that cannot be efficiently certified to be well-formed, even if this means excluding some inputs that are not in fact errors.

Thus inference for engineering automation of computer based systems is subject to very different constraints than the kind of inference that has been studied in mathematics and logic, and has different goals and priorities.

We use the term *lightweight inference* to denote inference systems that can operate within these constraints, which require soundness and extremely high efficiency but tolerate incompleteness and limited expressive power.

The rest of the paper is organized as follows. Section 2 discusses the requirements for the inference facilities needed to support engineering automation. Section 3 presents some past successes in automated inference and automated derivation on a large scale, and examines the factors that enabled these successes. Section 4 identifies some of the most promising past approaches that may grow into future solutions to the lightweight inference problem. Section 5 contains conclusions.

2 Inference Requirements for Engineering Automation

This section outlines the requirements that must be met by automated methods for lightweight inference.

1. *The methods must give reliable conclusions.* This is the soundness requirement identified above, which is essential for practical impact.
2. *The methods must be effective on a large scale.* This is the efficiency requirement, which depends on the context. The design aids supported by lightweight inference can be separated into two categories, immediate and background. Immediate feedback is intended to alert the designer to relevant design issues or faults as they are introduced. Response time is limited to a few seconds for immediate feedback, because otherwise the designer's attention will shift to different issues, and slow feedback will interrupt thought rather than aiding it. Background analysis tasks must take no longer than an overnight run to be practical. In either case, the inference mechanism must be completely automatic, without interactive guidance from a human user - otherwise the process would be too slow and too expensive to be cost effective. The interaction paradigm in this case is precomputation of anticipated queries whose results are displayed only when the programmer requests them. In either case, the efficiency requirements are more stringent than commonly assumed in theoretical work on inference.
3. *The methods must be able to solve problems that occur in practice.* Complete coverage is not required for practical usefulness. Special purpose methods that are limited to particular special cases are acceptable and may be desirable if they perform well on cases that arise in practice.
4. *The methods must be able to perform inferences "obvious" to people.* Due to the extreme efficiency constraints, we should not expect lightweight inference systems to solve deep mathematical problems that puzzle human experts. We should expect these systems to be able to determine properties that are obvious to professional engineers. The importance of automated inference is to be able to handle very large numbers of such problems, at much higher speeds and with much lower error rates than people could accomplish. The inference capabilities should handle the parts of the engineering process that appear conceptually routine when considered in isolation. Such issues can be major problems in engineering practice because of sheer volume of detail and large numbers of relatively simple but interacting constraints.
5. *The methods must be able to find "solutions" in addition to deciding properties.* Often engineers need examples or counter examples in addition to deciding whether given properties are true or false. Sometimes this problem arises in the form of determining particular values for some parameters that will make certain logical constraints true, sometimes with the additional goal of optimizing some objective function. A related problem is finding the weakest set of additional constraints that will

suffice to satisfy a given goal statement. In the context of software engineering, it is desirable to integrate inference capabilities with facilities for synthesizing programs or design artifacts expressed in other kinds of formal notations.

6. *The methods must be feasible for practicing engineers to learn and use.* This puts a premium on simple conceptual models of engineering processes and user interfaces that match the thinking habits of typical engineers. A successful strategy for simplifying the interfaces has been to encapsulate the relevant but subtle mathematical concepts inside of tools [16, 19, 24]. It is acceptable to require toolsmiths to have levels of mathematical skill that far exceed those of typical software engineers using tools with internal lightweight inference capabilities. Interface amenities such as active documentation, explanation, and guidance facilities can help, but they are no substitute for conceptual simplification at the interfaces and information hiding applied to deep theories of computing.
7. *The methods must have a failure interface to handle incompleteness.* Incomplete methods may fail. These cases must be explicitly reported as failures, so that there is no danger of basing delivered products on faulty conclusions. In such cases, the inference system should help isolate and diagnose the causes of failures, and provide guidance about how to mitigate or work around them. Such failures are often indications of particularly difficult parts of the problem. If the engineers are to solve the problems that the automated systems cannot handle, they will need assistance in isolating, simplifying and understanding them to have much chance of success.
8. *The methods should be robust and predictable.* The methods should terminate gracefully when they fail. For best acceptance by engineers, the cases in which the methods are expected to succeed and the amount of time they will require should be predictable. It is best if tractable special cases can be automatically detected by the system, and solution times estimated in advance, particularly if they are long. Since efficient methods tend to involve large numbers of rules, this puts a premium on computer aid for analyzing behavioral and performance properties of the rules.
9. *The methods must be adaptable.* Engineering automation is a complex problem domain that is not currently completely understood. In particular, an important part of the problem is productive interactions with skilled people and business organizations. Prototyping will be necessary to test the practical viability of research results in realistic engineering contexts, identify new issues implicit in those contexts, and improve automation support to address those issues and to provide gradually improving capabilities. Economic concerns also require some early demonstration of some returns on investment before all aspects of the problem are solved. This puts a premium on computer aid for changing and extending the rules that drive lightweight inference systems, and on improved models and representations for formulating and transforming the rules.

2.1 Some Automatable Engineering Tasks

Lightweight inference needs to be able to address problems that occur in practice but are not "too hard". To make this idea concrete, this section presents some examples of software engineering subtasks that should be completely automatable with the help of lightweight inference facilities and proper choice of safe approximations.

1. *Type inference.* Synthesis of type declarations in types programming or specification languages, and generation of diagnostics in case of type errors.
2. *Non-local references.* Resolving non-local references, synthesizing Ada WITH statements, and generating menu choices when there is more than one type consistent choice.
3. *Uninitialized variables.* Detecting references to uninitialized variables, and generating appropriate warning displays at design entry time.
4. *Exception handlers.* Determining the set of exceptions that can be raised by each statement, and synthesizing default exception handlers.

5. *Closing files.* Determining the set of open files and synthesizing *close* statements at the end of the appropriate scope.
6. *Locks.* Determining the set of locks needed and synthesizing statements to acquire and release locks as needed.
7. *Freeing storage.* Detecting local dynamically allocated objects and synthesizing storage recycling operations when safe.
8. *Slicing.* This is a form of dependency tracing that involves forming transitive closures. Applications include finding all program statements that can affect the truth of an invariant, identifying unreachable code, factoring unstructured descriptions into logically cohesive modules, and many others.
9. *Stubs.* Synthesizing stubs that enable execution, demonstration, or delivery of partially completed systems.
10. *Concrete interfaces.* Synthesizing default concrete interfaces, including graphical user interfaces, from abstract interfaces (such as those implicit in an object design or an essential model of the system).
11. *Test scaffolding.* Synthesizing the additional code needed to test an implementation module according to a given testing approach.

While each of these issues is in some sense routine, all of them consume substantial engineering time in practice, especially when error correction and proper response to modifications of engineering artifacts (requirements, specifications, decompositions, interfaces, programs, ...) are taken into account. Furthermore, each of them involves fairly sophisticated design and process considerations and non-trivial design decisions if it is to be resolved in a systematic way to meet stringent quality standards, in a realistic engineering environment where complete coverage is needed and common academic simplifying assumptions cannot be used.

2.2 Some Partially Automatable Engineering Tasks

This section presents some more difficult software engineering tasks that should be partially automatable with the help of lightweight inference facilities.

1. *Timing analysis.* For real-time applications, it is often necessary to get accurate upper bounds on how much time it will take to execute a given subprogram. Lightweight inference techniques should be able to automatically determine the number of microseconds per control block, and whether or not the recurrence relations that lead to bounds on the number of loop iterations and the depths of the recursions in the subprogram match any of the solution patterns in a database of known solutions. Interactive help or heavyweight inference support may be needed to solve recurrence relations that do not match the patterns in the database.
2. *Space analysis.* Embedded systems sometimes need accurate bounds on the amount of space needed to execute a given program. Lightweight inference techniques should be able to derive the number of bits per node or object. Interactive help may be needed to get bounds on the length of a linked list or the number of instances of a type used in the program that do not match the patterns in a solution database.
3. *Completing loops and data structures.* A mature engineering automation environment could include decision support facilities that can synthesize statements to restore invariants after manually designed code has changed some loop variables or data structure components. Lightweight inference should be able to automatically determine the sets of affected variables and weakest preconditions of sections of straight line code. Some interactive help may be needed for synthesizing the statements to restore the invariants based on this information for cases that cannot be handled by a database of synthesis patterns.

Note the common theme of a database of known and verified solutions that serves as an online handbook. Reuse of standard design patterns is a common approach in engineering that can benefit from automated decision support even if the process cannot be completely automated. The key is to automate the relatively routine parts of the process: this speeds up those parts and reduces the incidence of human errors, while reducing the intellectual load on the engineers, who can concentrate on the parts of the design that are not routine. In most applications the routine parts will account for a major fraction of the decisions in a large scale design, so that the proposed automation facilities will have substantial impact. This illustrates typical contexts where incomplete but reliable automatic methods are expected to be useful in engineering automation.

2.3 Design Representations and Characteristics

This section briefly characterizes the types of design representations used in software design to further characterize typical applications of lightweight inference.

Software designs involve many-to-many relations of many kinds. Graphs and hypergraphs are common, and appear in many different guises. For example, graphics are used for display, dependencies are used for synthesizing build procedures, data flow is used for optimization and slicing [5], and links are used for web navigation. These structures are typically combined with annotations in formal notations. Special purpose techniques for handling such structures are therefore valuable.

Software designs involve named objects with scope rules. Examples are variables, design modules, types, requirements, and so on. These names can typically be overloaded, so that some inference may be needed to resolve them. Names typically have many occurrences, which introduce dependencies of various kinds. Inference could be used to materialize dependencies explicitly, check consistency constraints, and derive change impact properties.

Software designs are typically updated concurrently by teams of designers, working in a distributed environment with networking. Inference could be used to check or maintain relations between different design artifacts or documents and alert team members to impending interactions with decisions made by other team members.

2.4 Common Types of Inferences

This section abstracts and summarizes the previous characterizations of typical applications of lightweight inference in engineering automation.

In engineering contexts it is often necessary to check or determine non-local properties of design objects. Dependency relations must be maintained and processed, and closure calculations of many kinds are commonly needed. Examples of dependencies include data flow dependencies, control flow dependencies, subprogram call graphs, subtype relations, requirements dependencies, and configuration dependencies between versions[18]. An example of a closure calculation is determining the set of exceptions that can be raised by a given subprogram. A typical non-local property is whether or not a subprogram or a variable is used by more than one concurrent thread.

There is often a need for large numbers of inferences that are conceptually simple when considered individually. For example, properties such as whether a given identifier denotes a predefined type arise often in program synthesis, because the applicability of synthesis rules depends on such properties. One at a time, they may appear trivial, but when they come by the millions in a large scale application, they can be a major problem. We need systematic and computer-aided methods for handling issues like this in many variations.

3 Past Successes of Automated Inference/Derivation

There is a substantial amount of past work relevant to the goals of lightweight inference, although much of it is weakly related to proofs and logic. This section briefly identifies and characterizes some of that work.

3.1 Some Examples

Here are some of the contexts where simple inferences have been successfully automated on a large scale.

1. *Optimizing compilers.* Compilers routinely determine properties of large programs to drive optimization processes and check for some kinds of semantic errors.
2. *Databases.* Queries on databases determine properties of large data collections.
3. *Symbolic mathematics.* Symbolic mathematics systems solve large math problems, some of which are beyond the practical capabilities of unaided humans.
4. *Optimization.* Optimization methods such as linear and integer programming find solutions to large problems. Another paper in this proceeding [15] gives an example of this relevant to software engineering automation.
5. *Model checkers.* Model checkers find problems in complex designs and protocols.
6. *Schedulers.* Real-time scheduling algorithms establish complex existence properties.
7. *Heuristic search.* Heuristic search methods find solutions in complex domains such as games (chess) and VLSI design (routing and layout).

The first four of these contexts are quite mature and many commercial tools are available. Based on this past experience, we conclude that lightweight inference should be feasible. However, the kinds of lightweight inference needed for engineering of computer based systems have not been intensively studied, and we believe that substantial progress in this area is possible.

3.2 Common Themes

Several past successes rely on domain-specific inference and derivation procedures. These procedures rely on special properties of the application domains to achieve their efficiency and effectiveness. Expressive power is commonly limited and tailored to the application domain. The methods produce accurate results when they succeed, and produce error messages when they fail. These error messages approximately explain failures, and although current facilities leave much to be desired in this area, the error messages often do contain enough information to enable skilled users to diagnose and correct the causes of the failures.

The past successes also have some common difficulties. Foremost among these is that the decision rules are complex and quite difficult to create, analyze, certify, and extend. Domain-specific inference often requires large numbers of similar rules. These systems are generally not very modular, and consequently they are very difficult to extend and refine. In many cases inference rules are implicitly encoded in complex algorithms. Even if the rules are explicit, they may not be systematically organized. Generalization is weakly supported, or not at all. There is little or no automated decision support for creating, analyzing, organizing, and improving the rules.

Another problem is that failure diagnosis is incomplete and sometimes inaccurate. Most systems do not produce advice on what to do to work around failures, and some do not even provide information that could help to localize the cause of the failure.

3.3 How Efficiency was Achieved

A principal challenge for lightweight inference is achieving adequate efficiency. This section briefly examines how past successes managed to get enough efficiency to scale up.

A major theme has been to avoid nondeterminism as much as possible. Since the cost of a search is typically exponential in the number of undetermined choices, many methods go to great lengths to reduce or eliminate choice points. Special structures of the problem domain are exploited to accomplish this, often via dominance properties, equivalences, special representations for entire classes of cases, and heuristics for pruning searches.

Another theme is to keep inference chains short. This leads to large numbers of very specific rules and algorithms that are coupled to the structure of the problem space.

Using memory to avoid recomputation and optimizing the frequently repeated steps are additional strategies that has been used to improve efficiency in the past successes.

4 Future Directions

This section identifies some of the most promising existing technologies and indicates aspects that could be improved for application to lightweight inference. These methods have been singled out because they have relatively good efficiency properties and they are applicable to relatively large domains. They also share the desirable properties of declarative rule representations and referential transparency.

4.1 Compiler Technology

Compilation is the most mature application of large scale inference for engineering of computer based systems. The main inference technology used in compilers is attribute grammars [14, 9]. This technology is mentioned first because it is the most mature and the most efficient.

The main strengths of this approach are efficient methods for evaluating attributes, efficient methods for updating them when a design changes, language-based structuring for the rules, and conceptual simplicity.

Efficient evaluation is achieved by making the rules completely deterministic and by using memory to store attribute values so that each attribute is evaluated at most once.

Efficient update is achieved by keeping track of dependencies and updating attribute values only when something they depend on has changed.

The rules are organized according to the structure of the source language. This helps in managing, debugging, and enhancing large sets of rules.

The approach has been widely applied to construction of compilers and other kinds of translators, and is easy for practicing software engineers to understand and use.

The main weaknesses of the approach are rule set complexity, weak support for abstraction and modularity, lack of support for refinement and fusion of decisions, lack of support for objects, generalization and collections, lack of precise semantics for scenarios where parts of the source are synthesized using computed attributes, unidirectional information propagation, bias towards text-based design representations due to the formulations based on syntax trees.

Large rule sets appear to be inherent in the determinism that produces the efficiency advantages of the approach. Our desire for improved modularity and higher conceptual levels of support in modeling the rules is motivated by the desire to apply the best known strategies for mitigating the conceptual complexity that stems from large rule sets with many interdependencies. This issue is on the critical path because rule sets will need to be even more complex than those in current compilers to provide the envisioned levels of engineering automation for computer based systems.

The use of computed attributes to synthesize routine parts of designs is at the heart of our goals for engineering automation. Foundational work is needed to develop formal systems that can provide a sound semantic basis for this kind of structure.

Current attribute grammar approaches support derivations that apply rules only in a single, statically determined direction. This helps in achieving high efficiency. However, design representations with consistency constraints that are automatically enforced via synthesis rules do not always naturally fit into a master/slave pattern. In situations where the necessary action is uniquely determined in either case, we would like to be able to update either end of a dependency chain and have the other end readjusted automatically to restore consistency.

Attribute grammars have their roots in a traditional text-based view of formal languages and syntax. Modern design uses a mixture of diagrams and text, both of which can have a precisely defined formal structure. It would be very useful to generalize attribute grammar ideas to include formal structures whose source form includes graphics as well as text [17].

4.2 Rewrite Rules

Another well studied context for automated inference is equational logic and rewrite rules [13, 10, 6, 7].

The strengths of this approach are its well developed theory, which can in principle support systematic analysis of rules, its type structure, and the associated (checkable) validity constraints on rules. This technique also has some efficiency benefits, which are somewhat weaker than the attribute grammar approach:

for rule sets with the church-rosser property, it is possible to replace search with deterministic choice of which rule to apply at each point without affecting the result.

The weaknesses of the approach are that typical engineers do not understand how to use it, it supports only functional (single-valued) attributes, it does not support shared design objects, change propagation and rule refinement are not supported, rule analysis may not be computable, links to engineering design representations are missing, and current implementations are not efficient enough to handle very large problems.

Our experience with trying to teach algebra and rewrite rules to master's students in computer science indicates that most of them fail to understand the principles deeply enough to be able to apply them in synthesis and engineering design. We believe that this population represents a reasonable upper bound on the skill levels of practicing software engineers. This skill requirement is a barrier to widespread application, which suggests that lightweight inference techniques based on this approach must hide the algebra inside a tool that provides simpler design representations to its users.

4.3 Resolution

Another well known approach to inference is Horn clause logic and resolution [23, 21].

The strengths of the approach are its well developed theory, which can also in principle support analysis of rules, its support for multi-valued relations, and conceptual simplicity. This approach is less efficient, since nondeterministic choice and backtracking are involved. However, it is a local optimum point with regard to efficiency because most general unifiers handle as large a class of cases in a single step as possible. Engineers have used Horn clause logic for many applications, although many of these have been in the guise of PROLOG and have relied on non-logical features that enable the use of an imperative rule style familiar from programming.

Some weaknesses of this approach are poor efficiency, lack of support for modules, generalization, and change propagation, difficulties in dealing with negative information, and possible masking of inference failures by the closed world assumption and non-logical constructs such as cut.

It is hard to integrate resolution with efficient deterministic procedures. Current strategies for achieving efficiency rely on imperative features and non-logical constructs, which destroy referential transparency and make rules hard to analyze. There do not appear to be good methods for integrating resolution with control heuristics. The control heuristics that exist are formulated in terms of implementation level concepts, or abstract theoretical concepts, such as orderings on function symbols. These are foreign to the engineers who will be using the systems, and are difficult to use due to lack of reliable and systematic guidelines for their use.

Negation as failure complicates the theory and also makes rules harder to analyze.

5 Conclusions

Lightweight inference is needed for engineering automation, including the promising reliable generator strategy for achieving cost-effective high quality software (see Section 1), and there is reason to believe that the needed capabilities are feasible. Some methods have been successfully applied (see section 3.1). However, none of the known methods is completely satisfactory. Application specific methods are needed for scalability, and better ways to develop such methods are needed.

Areas for future research related to lightweight inference include better models of inference rules and rule subsystems, better analysis capabilities for rules, better synthesis and transformation capabilities for rules, procedure synthesis capabilities for rule compilation, improved methods for maintaining inferences as hypotheses change, and parallel and distributed inference engines to support collaborative design and engineering decision fusion.

Cost-effective improvements in software quality are badly needed by society. The software engineering community would be well advised to demonstrate practical realization of such improvements relatively soon. We believe that the reliable generator strategy is a good way to do this. Research areas relevant to realizing the reliable generator strategy include improved formal models of requirements issues and dependencies; computable connections between requirements issues, software architectures, and program generation rules;

formal methods for developing, transforming, and certifying program generation rules; compilation of program generation rules into program generators; and integration of lightweight inference with formal models of program generation rules.

References

- [1] S. Badr, Luqi, Automation Support for Concurrent Software Engineering, *Proc. of the 6th International Conference Software Engineering and Knowledge Engineering*, Jurmala, Latvia, June 20-23, 1994, 46-53.
- [2] D. Berry, Formal Methods: The Very Idea, Some Thoughts About Why They Work When They Work, *Proc. of the Monterey Workshop on Engineering Automation for Computer Based Systems*, Carmel, CA, October, 1998.
- [3] V. Berzins, Luqi, An Introduction to the Specification Language Spec, *IEEE Software*, Vol. 7 No. 2, Mar 1990, pp. 74-84.
- [4] V. Berzins, Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley Publishing Company, 1991, ISBN 0-201-08004-4.
- [5] V. Berzins, *Software Merging and Slicing*, IEEE Computer Society Press Tutorial, 1995, ISBN 0-8186-6792-3.
- [6] H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 1: Equational and Initial Semantics*, Springer-Verlag, Berlin, 1985.
- [7] H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, Springer-Verlag, Berlin, 1990.
- [8] S. Fickas, Automating the Transformational Development of Software, *IEEE Transactions on Software Engineering*, Vol. 11 No. 11, Nov 1985, pp. 1268-1277.
- [9] R. Gray, V. Heuring, S. Levi, A. Sloane, W. Waite, Eli: A Complete, Flexible Compiler Construction System *CACM*, Vol. 35, No. 2, Feb. 1992, pp. 121-131.
- [10] J. Guttag, D. Kapur, D. Musser, Derived Pairs, Overlap Closures, and Rewrite Dominoes: New Tools for Analyzing Term Rewriting Systems, Eli: A Complete, Flexible Compiler Construction System *Automata, Languages, and Programming, Ninth Colloquium*, Lecture Notes in Computer Science, no. 140, Springer-Verlag, 1982, pp. 300-312.
- [11] M. Harn, V. Berzins, and Luqi, Software Evolution via Reusable Architecture, *Proceedings of 1999 IEEE Conference and Workshop on Engineering of Computer-Based Systems*, Nashville, Tennessee, March 1999, pp. 11-17.
- [12] E. Kant, On the Efficient Synthesis of Efficient Programs, *Artificial Intelligence*, Vol. 20 No. 3, May 1983, pp. 253-36. Also appears in [22], pp. 157-183.
- [13] D. Kapur, D. Musser, X. Nie, The Tecton Proof System: Introduction and User's Guide Institute for Programming and Logic, State University of New York, 1992.
- [14] D. Knuth, Semantics of Context-free Language *Math. Syst. Theory*, Vol. 2 No. 2, June 1968, pp. 127-145.
- [15] H. Kwak, I. Lee, O. Sokolsky, Parametric Approach to the Specification and Analysis of Real-Time System Designs based on ACSR-VP, *Proc. of the Monterey Workshop on Engineering Automation for Computer Based Systems*, Carmel, CA, October, 1998.
- [16] Luqi, M. Ketabchi, A Computer Aided Prototyping System, *IEEE Software*, Vol. 5 No. 2, Mar 1988, pp. 66-72.

- [17] Luqi, V. Berzins, R. Yeh, A Prototyping Language for Real-Time Software, *IEEE Transactions on Software Engineering*, Vol. 14 No. 10, Oct 1988, pp. 1409-1423.
- [18] Luqi, A Graph Model for Software Evolution, *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, pp. 917-927, Aug. 1990.
- [19] Luqi, ed., Special Issue on Computer Aided Prototyping, *Journal of Systems Integration*, Vol. 6, No. 1-2, 1996.
- [20] R. Lutz, Analyzing Software Requirements: Errors in Safety-Critical Embedded Systems, TR 92-27, Iowa State University, AUG 1992.
- [21] P. Padawitz *Computing in Horn Clause Theories*, Springer-Verlag, Berlin, 1988.
- [22] C. Rich, R. Waters, Eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.
- [23] J. Robinson, A Machine-oriented Logic Based on the Resolution Principle *JACM*, Vol. 12, 1965, pp. 23-41.
- [24] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood, The Deductive Composition of Astronomical Software Subroutine Libraries, *Proceedings of the twelfth international Conference on Automated Deduction*, 1994, pp. 341-355.

Reactive Verification with Queues

Nikolaj S. Bjørner*†‡§

Abstract

Temporal logic is the *right* specification language to capture requirements of reactive systems. To reason about specifications and implementations, that can in general be infinite state, STeP [3] (The Stanford Temporal Prover) offers verification rules and diagrams to decompose the verification task, generation of invariants to bootstrap verification, and finally integration of decision procedures to dispense with basic verification conditions. We here illustrate the scope of decision procedures for the data-type of queues in verifying a simple reactive controller.

Keywords: Decision procedures, temporal logic requirements, reactive systems.

1 A reactive control problem

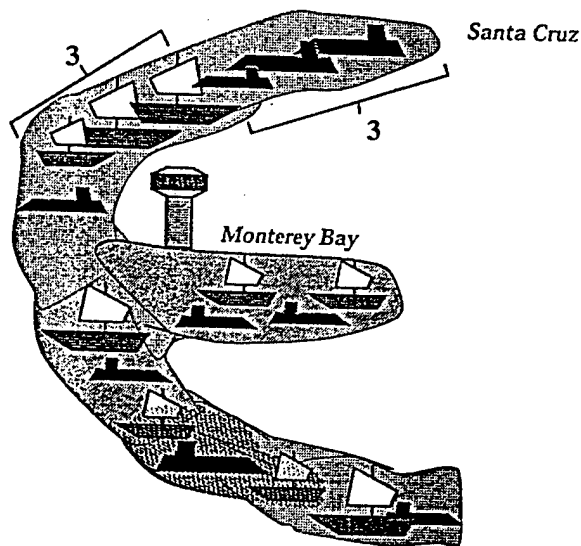


Figure 1: A network traffic controller

To understand the controller that we develop in this paper some familiarity with the Californian geography will be helpful. In particular we will assume that Santa Cruz is north of Monterey Bay. Assume also that the city council in Santa Cruz has decided that the boat traffic passing through Santa Cruz on its way from Monterey should be chunks of equally many submarines and sailboats alternating with a regular period. A frustrated council member asks Clint Eastwood in Carmel to sponsor a tower in Monterey Bay to control the flow of boats, at which Clint asks him, voice rasping, "Are you feeling lucky today, punk?" But that is not our main point. Figure 1 illustrates a possible scenario of the intended flow: boats and submarines from the south enter the bay in any order, the control tower has the option to keep the incoming vessels in the bay for a while or send them up north, but only in such a way that submarines and boats alternate with a regular period of length 3.

As with many other political forums, the council is bitterly divided in Santa Cruz, and is unable to reach a bipartisan agreement of whether the length of the period should be 3 or 5. In the present assembly those favoring a period of length 3 has the majority. Since the council

is soon up for re-election it does everything to make sure that popular demand is satisfied, also should the new council have a majority for a period of length 5. A design constraint for the controller is therefore that it should be easily reconfigurable to whatever period N of alternation that fits any assembly.

*This work was supported in part by ARO under grants DAAH04-95-1-0317, the National Science Foundation under grants CCR-95-27927 and CCR-98-04100, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, a gift from Intel, DAAH04-96-1-0122 and DAAG55-98-1-0471, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

†E-mail address: bjorner@kestrel.edu

‡The author thanks Zohar Manna and the STeP group for a stimulating working environment,

§as well as the organizers of the workshop on *Engineering Automation for Computer Based Systems*.

Since Clint cannot always sit in the control tower we are faced with the problem to design a controller program that directs and redirects boats and submarines to the bay or up north such that the period of alternation is N .

2 Requirements

While pictures may give some intuition as to what sort of controller is to be designed they cannot directly be used to describe the behavior of the controller over time. For this purpose we use linear time temporal logic because it allows to capture requirements from an observer's point of view, directly appeals to state changes over time, and is supported by an adequate supply of model checking algorithms for finite state systems and verification rules for infinite state systems. In linear time temporal logic a formula is evaluated over a computation sequence on the environment's and system's influence on observable variables.

Linear time temporal logic allows for instance to specify that

1. Ships and submarines *do not* alternate within distance N . Thus, for every sequence of at most N vessels sent up north, if the first and last vessel in the sequence are of the same type, then every vessel in the sequence is of the same type.
2. Ships and submarines *do* alternate on distance $N + 1$. Thus, for every sequence of $N + 1$ vessels sent up north, the first and last vessel in the sequence are of different types.
3. Every vessel sent up north came from south.
4. The controller reacts on sufficient stimulus. Thus, if infinitely many submarines and ships have been sent up from south, then infinitely many submarines and ships are sent up north.

Linear time temporal logic allows to capture these requirements leaving fewer ambiguities. This formalization occupation alone is in fact one of the most rewarding components of a system design. For instance, it may reveal that the above informal requirements leave open whether the controller is allowed to delay vessels of its choice arbitrarily.

However, the focus in this paper is on verification support for checking selected specifications against proposed implementations. We will therefore limit ourselves to the formalization of the two first properties which are invariants. Invariants can be specified with the form $\Box \varphi$, where φ is a first-order formula, and $\Box$ is the *always* temporal operator. The judgment $S \models \Box \varphi$ asserts that the formula φ holds on every reachable state of system S .

3 Queues

A closer study of the proposed controller's environment reveal that the queue data-type can be used with advantage to model the flow and temporary storage of vessels.

3.1 Monterey Bay

To properly preserve the wild-life and not disturb the singing (and occasionally doped) amateur pilots in Monterey Bay incoming vessels are stored in the bay in a last-in first-out order. A suitable abstraction of the bay is therefore a *stack*, or equivalently a *list*. Lists have standard constructors ϵ (for the empty list), *cons* to add an element, and selectors *head* and *tail*. With suitable donations from the Packard Foundation, we can always extend the size of the bay and therefore assume it has infinite capacity.

3.2 The coastal Mother Road¹

The flow of vessels is on the other hand more naturally modeled using *channels*. Here the primitive operations consist in appending items to the end of a channel, and taking items out from the front, thus processing

¹To continue: "66 is the mother road, the road of flight." John Steinbeck, *The Grapes of Wrath*.

elements in a first-in first-out order. In the programming language SPL which serves as one of possible ways to present reactive systems to STeP the statement

$$consumer \Leftarrow v$$

has as effect to put v in the end of channel *consumer*. The statement

$$producer \Rightarrow x$$

can be executed when the channel *producer* is non-empty, and has the effect of removing the first element from *producer* and updating x to it.

3.3 Queues as a common denominator

We can kill two birds with one golf-ball (on Pebble Beach) by modeling both lists and channels using queues. Queues are lists with pairs of symmetric constructors and selectors respectively. Thus, symmetric to the constructor *cons* there is a constructor *revcons*, which appends an element to the end of a list instead of to the front as *cons* does. Similarly, symmetric to *head* there is a selector *last*. The functionality of selectors and constructors is illustrated in Figure 2.

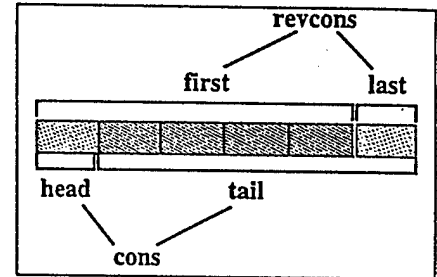


Figure 2: Queue constructors and selectors

3.4 Relations over queues

The most basic relation over queues is undoubtedly equality between these. Equality is however in no way the only relation one may want to express between queues. For example, the test whether a queue q contains element e , suggestively written $e \in q$, is another natural candidate. More generally one can assert sub-queue relationships using $q \preceq r$ to assert that queue q is a sub-queue of r . Special cases are when q is a prefix respective suffix of r . These relations are illustrated in Figure 3.

3.5 Decision procedures for queues

Automatic support for queues in quantifier-free formulas with the presented vocabulary has been developed in [2]. A result from this work that we will make use of here is a decision procedure for validity, or equivalently satisfiability, for quantifier free formulas, where terms are built using variables, together with operations

$\epsilon, \text{head}, \text{tail}, \text{cons}, \text{first}, \text{last}, \text{revcons},$

and atomic formulas are built using basic relations

$e \in q, q = r, q \preceq r, \text{prefix}(q, r), \text{suffix}(q, r) .$

Moreover, the proposed decision procedure can be integrated (tightly) with solvers for other theories, such as arithmetic, to provide a combined validity checker for a union of several theories. Space limitations prevent us from recreating all details here, but readers familiar with interactive verification may appreciate what

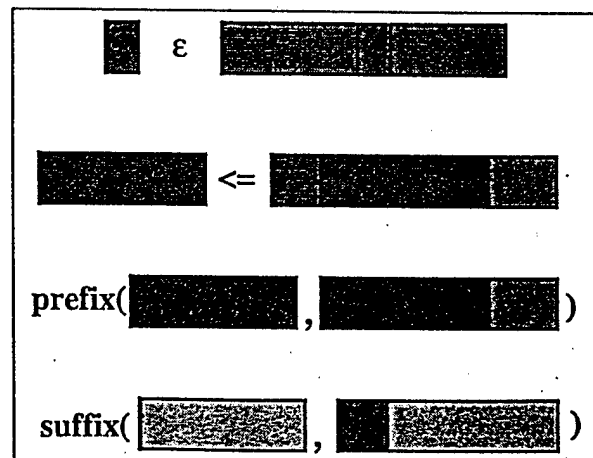


Figure 3: Relations among queues supported by decision procedures

relief decision procedures give. For example, we can use and reason directly about $\text{revcons}(l, a)$ instead of defining it indirectly:

$$\begin{aligned}\text{revcons}(l, a) &= \text{rev}(\text{cons}(a, \text{rev}(l))) \\ \text{rev}(l) &= \text{rev2}(l, \epsilon) \\ \text{rev2}(\epsilon, l) &= l \\ \text{rev2}(\text{cons}(a, l), m) &= \text{rev2}(l, \text{cons}(a, m))\end{aligned}$$

In this case we would often have to rely on heuristic support for inductive reasoning to unwind the recursive definition of rev2 .

4 Implementation

Figure 4 suggests an implementation of the controller. It uses a stack to keep track of vessels that cannot be sent north immediately, a counter i to maintain how many vessels of the same value have been sent, and a flag turn to record whose turn it is. Symptomatic to the diseases of low-level programming we have used booleans to encode the distinction between sailboats and submarines. So bear in mind that *true* is shorthand for sailboat, and *false* is shorthand for submarine.

```

const N      : [1..]
in  producer : channel[1..] of boolean where producer =  $\epsilon$ 
out  consumer : channel[1..] of boolean where consumer =  $\epsilon$ 
local stack  : boolean list where stack =  $\epsilon$ 
local turn   : boolean where  $\neg \text{turn}$ 
local i      : integer where  $i = 0$ 

procedure emit(v) =  $\left[ \begin{array}{l} \text{consumer} \leftarrow v; \\ i := \text{if } i + 1 \geq N \text{ then } 0 \text{ else } i + 1; \\ \text{turn} := \text{if } i = 0 \text{ then } \neg \text{turn} \text{ else } \text{turn} \end{array} \right]$ 

Producer ::  $\left[ \begin{array}{l} \text{loop forever do} \\ \quad [\text{producer} \leftarrow \text{false or producer} \leftarrow \text{true}] \end{array} \right]$ 
||
Router ::  $\left[ \begin{array}{l} \text{local } x : \text{boolean} \\ \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_0: \text{if head(stack) = turn} \wedge \epsilon \neq \text{stack} \\ \text{then} \\ \quad \left[ \begin{array}{l} \ell_1: \text{emit(head(stack))} \\ \ell_2: \text{stack} := \text{tail(stack)} \end{array} \right] \\ \text{else} \\ \quad \left[ \begin{array}{l} \ell_3: \text{producer} \Rightarrow x; \\ \ell_4: \text{if } x = \text{turn} \\ \quad \text{then } \ell_5: \text{emit}(x) \\ \quad \text{else } \ell_6: \text{stack} := \text{cons}(x, \text{stack}) \end{array} \right] \end{array} \right] \end{array} \right]$ 

```

Figure 4: CONTROLLER

The implementation is described in a simple programming language with primitives for concurrency and communication via channels. It is one of the possible formats for presenting reactive systems to STeP. STeP compiles the program text into a *transition system*, which models each basic statement of the program as a transition. The transitions (atomic actions) are described using binary relations between the present and next state values of the system's state variables. For example, the statement ℓ_3 is enabled when the control counter ℓ reaches 3 and the channel *producer* is non-empty. It updates the control to $\ell := 4$ and dequeues

the first element of *producer* placing it in the variable *x*. Other system variables are unchanged. Described as a first-order relation the transition relation reads:

$$\begin{aligned} & \ell = 3 \wedge \text{producer} \neq \epsilon \\ \wedge & \ell' = 4 \wedge \text{producer}' = \text{tail}(\text{producer}) \wedge x' = \text{head}(\text{producer}) \\ \wedge & \text{consumer}' = \text{consumer} \wedge i' = i \wedge \text{stack}' = \text{stack} \end{aligned}$$

Fortunately, STeP provides translations like these automatically and behind the scenes. We use $\mathcal{T}$ to summarize the transitions of system S , τ to access individual transitions, and ρ_τ for the first-order transition relation associated with τ . The initial states of S are captured using a first-order formula Θ . For our program Θ is

$$\text{producer} = \epsilon \wedge \text{consumer} = \epsilon \wedge \text{stack} = \epsilon \wedge \text{turn} = \text{false} \wedge i = 0 \wedge \ell = 0$$

5 Deductive verification

The deductive methods of STeP verify temporal properties of systems by means of verification rules and verification diagrams. *Verification rules* reduce temporal properties of systems to first-order verification conditions [8]. The most widely used verification rule is (the essentially non-temporal) INV given in Figure 5. It reduces the verification of the invariant $\Box p$ to the first-order verification conditions in premises B1 and

For assertion p , B1. $\Theta \rightarrow p$ B2. $\{p\} \mathcal{T} \{p\}$ $S \models \Box p$
--

Figure 5: Basic invariance rule INV.

B2. The condition B2 is shorthand for

$$\bigwedge_{\tau \in \mathcal{T}} (p(\bar{x}) \wedge \rho_\tau(\bar{x}, \bar{x}') \rightarrow p(\bar{x}')) .$$

6 Generation of invariants

It is a property of the implementation that the *stack* variable contains only bits of the same value. The alert reader will notice that Figure 1 is misleading in this respect, but keep in mind that we are only analyzing one possible realization of the requirements. We can check the property by postulating the invariant:

$$\varphi_1 : \quad \Box ((\neg \text{head}(\text{stack})) \notin \text{stack}) \tag{1}$$

Note that if the *stack* is empty, then the invariant holds trivially although we have left the effect of *head* under specified in this case (not undefined as the programming language centric convention is).

The invariant φ_1 is not inductive, but by using the auxiliary invariants

$$\varphi_2 : \quad \text{at}_{\ell_{3,4}} \wedge (\text{head}(\text{stack}) \leftrightarrow \text{turn}) \Rightarrow \text{stack} = \epsilon \tag{2}$$

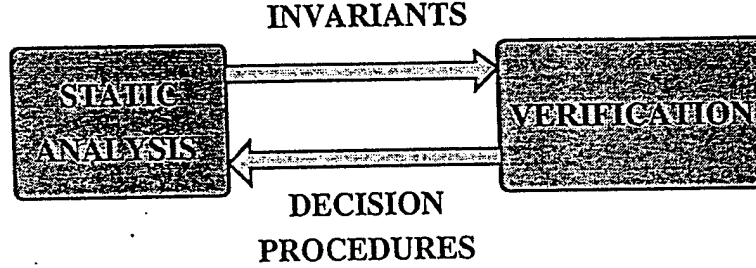
$$\varphi_3 : \quad \text{at}_{\ell_{5,6}} \Rightarrow (x \leftrightarrow \text{turn}) \wedge ((\text{head}(\text{stack}) \leftrightarrow \text{turn}) \rightarrow \text{stack} = \epsilon) \tag{3}$$

we can use rule INV from Figure 5 and the decision procedures for queues to automatically prove the property. We used the shorthand $\text{at}_{\ell_{3,4}}$ in place of $\ell = 3 \vee \ell = 4$, and $(\dots \Rightarrow \dots)$ as shorthand for $\Box(\dots \rightarrow \dots)$.

The invariants φ_2 and φ_3 are generated automatically by STeP and need therefore not be formulated independently by the practicing verifier. Generation of invariants and auxiliary assertions [4] is one of the features STeP provides to bootstrap deductive verification. In this case STeP's user only had to assert the main property to be verified, press a button to generate invariants, and invoke INV to let the decision procedures handle the sub-goals automatically.

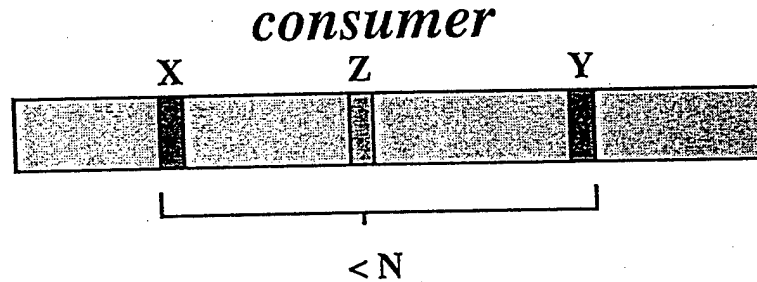
6.1 Abstraction

Invariant generation and decision procedures play an intimate role in an emerging practice in automatic verification of infinite state systems using abstraction [1, 5]. Invariants are generated from abstraction, which relies on techniques from static program analysis. Decision procedures, on the other hand, can be used to generate abstractions preserving more properties as they give more expressive power to the static analyzer. The coupling is loosely motivated by the picture below.



7 Verification

Suppose now that we wish to express that the bits in the consumer do not change value within distance N . Pictorially, if x and y are the same in *consumer*, and the distance between x and y does not exceed N , then any z between x and y must have the same value.



Using a sub-queue relation symbol $\preceq$, operations *head*, and *last*, which pick first and last elements in a queue, and a length measure $| \cdot |$ we can express this concisely using the invariant:

$$(\forall s) \left(\left(\begin{array}{l} s \preceq \text{consumer} \\ \wedge \quad 1 \leq |s| \leq N \\ \wedge \quad \text{head}(s) = \text{last}(s) \end{array} \right) \Rightarrow (\neg \text{head}(s)) \notin s \right) \quad (4)$$

The invariant is unfortunately not inductive, but can be established using the auxiliary invariants below. The predicate *suffix* states that s is a suffix of the queue *consumer*.

$$\Box (0 \leq i < N) \quad (5)$$

$$i > 0 \Rightarrow \text{last}(\text{consumer}) = \text{turn} \quad (6)$$

$$i = 0 \wedge \text{consumer} \neq \epsilon \Rightarrow \text{last}(\text{consumer}) \neq \text{turn} \quad (7)$$

$$(\forall s) \left(\left(\text{suffix}(s, \text{consumer}) \wedge 1 \leq |s| \leq N \right) \Rightarrow \text{if } |s| \leq i \text{ then } \neg \text{turn} \notin s \text{ else } (i = 0 \rightarrow \text{turn} \notin s) \right) \quad (8)$$

The good news is on the other hand that verification of both the auxiliary invariants and the main specification proceeds practically automatically thanks to the decision procedures for queues that we develop

in the following. The verification condition below is one of the proof-obligations that is produced as a sub-goal by the rule INV and established automatically using the decision procedures.

$$\left(\begin{array}{l}
 (0 \leq i \wedge i < N) \\
 \wedge (0 < i \rightarrow \text{last}(\text{consumer}) = \text{turn}) \\
 \wedge \left(\begin{array}{l} i = 0 \wedge \neg(\text{consumer} = \epsilon) \rightarrow \\ \neg(\text{last}(\text{consumer}) = \text{turn}) \end{array} \right) \\
 \wedge \left(\begin{array}{l} \text{suffix}(\text{first}(s), \text{consumer}) \\ \wedge 1 \leq |\text{first}(s)| \wedge |\text{first}(s)| \leq N \\ \rightarrow \\ \text{if } |\text{first}(s)| \leq i \\ \text{then } (\neg \text{turn}) \notin \text{first}(s) \\ \text{else } (i = 0 \rightarrow \text{turn} \notin \text{first}(s)) \end{array} \right) \\
 \wedge \left(\begin{array}{l} s \preceq \text{consumer} \wedge 1 \leq |s| \wedge |s| \leq N \rightarrow \\ \text{head}(s) = \text{last}(s) \rightarrow (\neg \text{head}(s)) \notin s \end{array} \right) \\
 \wedge \text{head}(s) = \text{last}(s) \\
 \wedge s \preceq \text{revcons}(\text{consumer}, \text{turn}) \\
 \wedge 1 \leq |s| \wedge |s| \leq N \\
 \wedge \left(\begin{array}{l} \text{first}(s) \preceq \text{consumer} \wedge \\ 1 \leq |\text{first}(s)| \wedge |\text{first}(s)| \leq N \rightarrow \\ \text{head}(\text{first}(s)) = \text{last}(\text{first}(s)) \rightarrow \\ (\neg \text{head}(\text{first}(s))) \notin \text{first}(s) \end{array} \right)
 \end{array} \right) \rightarrow (\neg \text{head}(s)) \notin s$$

Finally, we can verify that elements in distance $N+1$ in the *consumer* are always different using the auxiliary invariant (9) in establishing (10).

$$(\forall s) (\text{suffix}(s, \text{consumer}) \wedge i < |s| \leq N \Rightarrow \text{head}(s) \neq \text{last}(s)) \quad (9)$$

$$(\forall s) (s \preceq \text{consumer} \wedge |s| = N+1 \Rightarrow \text{head}(s) \neq \text{last}(s)) \quad (10)$$

8 Conclusions

So, what was the point? Did we tire the reader with yet another trivial example (albeit not a railroad crossing problem) to discredit formal methods, or at least our version of it? Did we use the *right* way for the requirements capture, as postulated in the abstract? Did we blaze our own trail to distinguish ourselves, and what can be reused by the community that does not subscribe to temporal logic based deductive verification.

My first purpose has naturally been to describe one of the ways to verify systems with STeP, and in particular highlight decision procedures which is related to my own technical interests. Temporal verification in this flavor consisted of several components: a language for capturing requirements, a language and underlying model for describing systems, and a verification methodology consisting of verification rules, generation of invariants, and decision procedures. The example has been chosen carefully such that standard model checking techniques could not be applied due to the non-fixed parameter N . Deductive methods are largely insensitive to such generalization.

But if one is not interested in verification of temporal requirements of a given system, what do we have to offer? Certainly the decision procedures for queues can handle data-domains independently of whether we are addressing properties of reactive systems, functional programs, or high-level specifications. The decision procedures for queues have even been used in an extensive benchmark for component based software retrieval [6]. Decision procedures offer the algorithmic counterpart of inference from first principles, which impedes the use of any verification technology about anywhere. Decision procedures also often terminate

quickly on invalid goals, at least the ones I have had the chance to test. This is particularly helpful in debugging the verification of a specification. Finally, decision procedures, even when implemented only incompletely, is one of the ingredients that can be used for something more fancy than the standard debugging and code optimizing environments for software. If the aim is to program with constraints, decision procedures are again the required core technology.

In summary, I would like to emphasize two points in connection with this paper: First, formalization, which was done only roughly in Section 2, is certainly essential for the documentation and success of the development of a software based system. "See source code for documentation", or "Its all in my mind", are unfortunately not hypothetical scenarios - to the authors own regretful experience. Adapting an independent formalization process allows to communicate and document system requirements and architecture. Formalization should naturally only be carried out to the extent that it contributes to the design and analysis of the constructed system. On the other hand, constructing a system so that it is amenable to formalizable analysis has never been a disadvantage (for me). Second, automatic support of high-level formalizable frameworks adds value to the development of software. This may be in the form of compilers for high-level programming languages (ML, SeqL), constraint-based programming environments, or expressive debugging tools, such as, deductive verification, extended static checkers, and model checkers. The technology for the latter is reaching a stage today where they can be used with some benefit in the design cycle along with standard debugging and development tools. Naturally, decision procedures play a central part of this process.

References

- [1] BENSALEM, S., LAKHNECH, Y., AND OWRE, S. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [7], pp. 319-331.
- [2] BJØRNER, N. S. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Computer Science Department, Stanford University, Nov. 1998.
- [3] BJØRNER, N. S., BROWNE, A., CHANG, E. S., COLÓN, M., KAPUR, A., MANNA, Z., SIPMA, H. B., AND URIBE, T. E. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proc. 8th Intl. Conference on Computer Aided Verification* (July 1996), R. Alur and T. A. Henzinger, Eds., vol. 1102 of LNCS, Springer-Verlag, pp. 415-418.
- [4] BJØRNER, N. S., BROWNE, A., AND MANNA, Z. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science* 173, 1 (Feb. 1997), 49-87. Preliminary version appeared in *1st Intl. Conf. on Principles and Practice of Constraint Programming*, vol. 976 of LNCS, pp. 589-623, Springer-Verlag, 1995.
- [5] COLÓN, M. A., AND URIBE, T. E. Generating finite-state abstractions of reactive systems using decision procedures. In Hu and Vardi [7], pp. 293-304.
- [6] FISCHER, B. *Automatic software retrieval*. PhD thesis, TU Braunschweig, 1998.
- [7] HU, A. J., AND VARDI, M. Y., Eds. *Proc. 10th Intl. Conference on Computer Aided Verification* (June 1998), vol. 1427 of LNCS, Springer-Verlag.
- [8] MANNA, Z., AND PNUELI, A. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

Generic Tools for Verifying Concurrent Systems*

Rance Cleaveland
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
USA
rance@cs.sunysb.edu

Steven T. Sims
Naval Research Laboratory
Code 5540
4555 Overlook Ave., SW
Washington, DC 20375-5337
USA
sims@itd.nrl.navy.mil

Abstract

Despite the enormous strides made in automatic verification technology over the past decade and a half, tools such as model checkers remain relatively underused in the development of software. One reason for this is that the bewildering array of specification and verification formalisms complicates the development and adoption by users of relevant tool support. This paper proposes a remedy to this state of affairs in the case of finite-state concurrent systems by describing an approach to developing customizable yet efficient verification tools.

1 Introduction

The field of automatic verification of finite-state systems has experienced tremendous advances over the past decade and a half, as efficient verification algorithms have been developed and associated tools built and applied to case studies of substantial complexity [13, 18]. Within the hardware community, commercial interest in these tools has even begun, as companies such as Intel, National Semiconductor and Chrysalis Symbolic Design have incorporated the use of automatic verification tools in their design processes. Despite these developments, however, verification technology remains largely unused in the software community in general, even in areas such as process control and communications protocols, where finite-state models form the basis of system implementations, thereby rendering them candidates for automatic analysis.

One may identify several cultural and technical reasons for this lack of uptake within the software community: unavailability of training, uncertainty about how to deploy formal analysis in the software process, skepticism about the benefits versus the costs of formal

*Research supported by ARO grant P-38682-MA, AFOSR grant F49620-95-1-0508, NSF Young Investigator Award CCR-9257963, NSF grant CCR-9505562, NSF grant CCR-9996086, and NSF grant INT-9603441.

analysis, etc. While it is beyond the scope of this paper to discuss all these concerns, we do note that even users who might be interested in formal approaches to the analysis of finite-state concurrent systems are confronted with the following issues.

1. Which design notation should be used for representing software artifacts? The literature contains a number of proposals, including Esterel [1], Statecharts [24], SDL [9], LOTOS [29], and CSP [25], to name only a few of the best-known ones.
2. How should requirements for designs be formulated? Again, the literature contains numerous suggestions, including finite-state machines, Computation Tree Logic [12] and Linear Temporal Logic [27], to name a few.

This bewildering array of choices has two negative consequences. The first is that no specification formalism has yet achieved a "critical mass" of users. The second is that tool support (necessary for any serious use of formal analysis) remains fairly primitive from a user's perspective; the fact that no "market" exists for any single formalism has dissuaded tool builders from expending the resources needed to build sophisticated and usable tools. The lack of appropriate tool support has in turn retarded the uptake of automatic verification among software designers.

In this paper we propose a framework for developing *generic* and *customizable* verification tools and investigate its use as a basis for efficient automated analysis of finite-state systems. The framework is intended to ease dramatically the task of developing usable tools for (operationally based) verification formalisms, thereby removing, at least in principle, one obstacle to the increased adoption of verification technology in practice.

2 Fundamental Concepts in the Verification of Finite-State Systems

This section sketches the concepts we deem fundamental for the analysis and verification of finite-state systems. The first two involve approaches to establishing that finite-state systems satisfy their specifications. In general, one may identify two schools of thought regarding the verification of systems: *logic-based* approaches and *refinement-based* techniques. The former typically involve the use of a temporal logic for describing desired system properties; one then uses a *model checker* to determine whether or not the properties hold of a putative implementation. The latter uses abstract, "high-level" systems as specifications; one then proves an implementation correct by showing that it "refines" such a specification (i.e. is related to it by an appropriate behavioral equivalence or preorder). Both approaches have their uses, and a number of temporal logics and behavioral relations have been proposed for verification purposes [13, 18].

So what is fundamental to these approaches? In the case of model checking, we and others [22, 3] posit that the modal μ -calculus [26] constitutes an expressive and efficient basis. This logic provides simple modalities and propositional constructs together with mechanisms for defining properties recursively. Efficient model-checking algorithms have been developed for fragments of this logic [2, 19, 21], and other temporal logics have efficient

translations into these fragments [3, 22]. Regarding refinement-based approaches, we argue for the fundamentality of (bi)simulation. Efficient algorithms exist for determining whether systems are related by bisimulation equivalence (simulation preorder) [30, 23, 11], and other relations may be computed efficiently by combining decision procedures for (bi)simulation with appropriate transformations on the underlying finite-state systems [10, 16]. In addition, general theories of bisimulation-based “diagnostic information” that explain why a system fails to refine another have been developed [11].

The final fundamental notion involves the definition of design notations for representing finite-state systems. In order to be usable as a basis for formal analysis, such notations must, in addition to having useful constructs, be equipped with a formal semantics that unambiguously defines an association between “programs” in the language and finite-state machines representing their behavior. To give such a semantics, we advocate the use of operational semantic in general, and structural operational semantics (SOS) [31] in particular, as a rigorous yet conceptually clear presentation style. SOS presentations consist of collections of inference rules that specify the single-step transitions of systems in terms of the execution steps of their components. Languages such as CCS [28], LOTOS [29] and CSP [25] have such a semantics, and it has become the preferred style for defining the meaning of constructs in process algebra [5]. An additional virtue of operational semantics, and SOS in particular, involves its connection with *simulation*: an operational account of a language implicitly defines how to simulate “programs” in the language.

3 The Concurrency Workbench And Analytical Gener- icity

The previous section presented a proposed foundation for the automatic verification of finite-state systems. In order for this theory to be of practical as well as theoretical value, one must show that it can be used as a basis for the development of usable verification tools. This section and the one following explore this issue by describing our experience with two associated automatic verification tools: the Concurrency Workbench of North Carolina [17] and the Process Algebra Compiler of North Carolina [15].

The Concurrency Workbench was originally conceived as a “laboratory” for experimenting with different techniques for verifying finite-state systems represented in CCS [16]. The tool incorporated implementations of bisimulation, prebisimulation and mu-calculus model-checking algorithms and provided support for easily customizing these algorithms to calculate a variety of different behavioral relations and for introducing new temporal constructs. The original public release of the system suffered from several performance bottlenecks, and consequently while it was easy to customize it could be frustratingly inefficient. The tool was nevertheless used successfully in the analysis of several case studies [7].

The Concurrency Workbench of North Carolina (CWB-NC) [17] represents a completely re-implemented version of the original CWB. Our goal in this effort has been to show that the inefficiencies of the CWB were due not to its genericity (as some have suggested) but rather to lower-level implementation issues that can be addressed in a design-language- and analysis-independent manner. Consequently, the CWB-NC retains the (pre)bisimulation/mu-calculus

orientation of the original CWB, but it contains more efficient implementations of the low-level routines. It also cleanly separates routines that are design-language-specific (parsers, unparsers, transition calculation) from those that are independent of the design notation (bisimulation, model checking, simulator) in order to facilitate modifications to the language that is supported. Figure 1 contains a representation of the architecture of the CWB-NC. The CWB-NC has been publicly available since September of 1996 from URL www4.ncsu.edu/~rance/WWW/cwb-nc.html and has been used in the analysis of several reasonably sophisticated case studies [14, 20]. While a detailed comparison has not been conducted, preliminary evidence suggests that the CWB-NC is 2-3 orders of magnitude faster than an earlier version of the CWB (specifically, version 6.0).

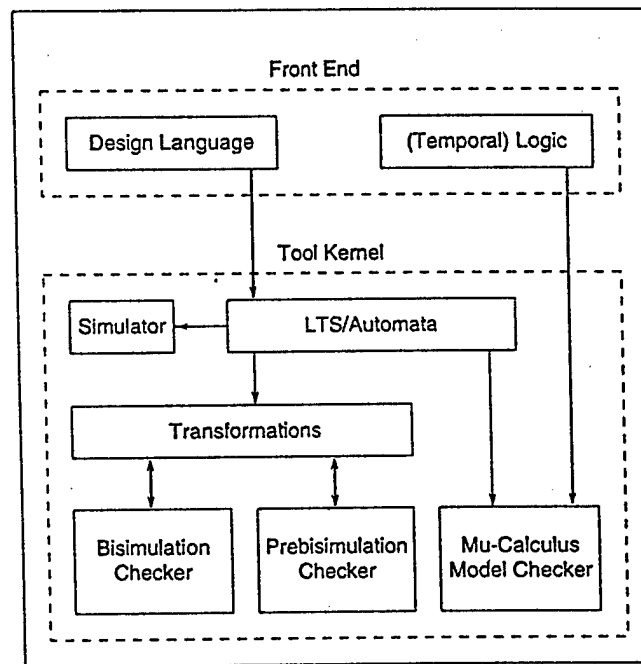


Figure 1: The Architecture of the CWB-NC.

4 The Process Algebra Compiler and Language Gener- icity

Our experience with the CWB and CWB-NC suggests that (pre)bisimulations and the modal mu-calculus form an efficient yet easily customizable basis for system verification. However, changing the design language supported by the CWB-NC requires substantial and delicate recoding in order for performance to be acceptable. In order to alleviate the difficulty of this task, the Process Algebra Compiler (PAC) project between NCSU and INRIA-Sophia Antipolis was undertaken with support from the US National Science Foundation and INRIA [15]. The PAC aims to produce efficient front ends for verification tools from high-level descriptions of the syntax and semantics of the design language the front-end is intended

to support. The Process Algebra Compiler of North Carolina (PAC-NC) constitutes the specialization of the PAC for the CWB-NC.

The PAC-NC takes as input files defining the abstract and concrete syntax of a design notation and its operational semantics as SOS rules and generates SML code (the implementation language of the CWB-NC) implementing parsers, unparsers and relevant semantic routines (primarily a transition calculator). A user may then insert these routines into the CWB-NC in order to change the design notation supported by the tool. Figure 2 graphically depicts this process. It should be noted that all versions of the PAC, including the PAC-NC, use the same PAC front end; they differ only in the code they produce, since different verification tools expect routines in different languages and with different functionalities.

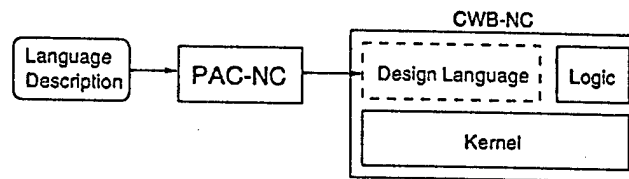


Figure 2: The PAC-NC Architecture.

Efficiency Issues. The CWB-NC makes extremely heavy use of the semantic routines for a design language; to construct an automaton from a design language “program”, for instance, the transitions function must be called for each state. Consequently, in order for the PAC-generated front ends to be usable, great care must be taken to ensure the efficiency of the automatically-generated semantic routines. To achieve this, the PAC-NC combines a general pattern-matching-oriented approach with two low-level optimizations in the semantic routines it generates. We briefly describe these here; the interested reader is referred to [15] for a more detailed account.

To build a function for a semantic routine from an SOS specification, the PAC-NC first analyzes the rules on the basis of the design language constructs they are applicable to. It then generates a function that, given a “program” in the design language, determines the applicable rules, recursively calculates the semantic information for appropriate subprograms, and then uses the rules to combine the results of the recursive calls appropriately. To make this process as efficient as possible, the produced routine also does the following.

Call caching. The results of certain previous recursive calls are stored in a table in order to avoid duplication of effort. (Which call results are cached in this manner is presently left up to the user, although this information could also be determined by analyzing the SOS rules appropriately.)

Tree flattening. Parse trees are represented in a compact fashion in order to facilitate hashing and equality-checking on trees.

Tables 1 and 2 contain the time and space results of some experiments with different PAC-generated front ends for the CWB-NC. The experiments were conducted on a 180 MHz Pentium Pro machine with 64 MB of memory. The timing table records the time needed to

construct an automaton from a given program, while the space table indicates the maximum heap size needed. The programs used include the following.

802-2: A simplified version of the IEEE 802-2 token ring protocol.

Mailer: A version of the electronic mail protocol used by the Computer Science Department of the University of Edinburgh in the late 1980s [6].

ATM: An account of version 3.0 of the User/Network Interface in the ATM communications protocol [10].

Emitter: A sender in a communications protocol.

Railway (LOTOS): A railway signaling scheme.

Railway (PCCS): The same railway signaling scheme in the PCCS process algebra [14].

In general, caching and tree flattening lead to significant improvements in timing behavior. Somewhat surprisingly, they also induce improved memory performance on occasion. This seeming anomaly results from sharing in the parse tree representations that caching in particular supports. It should also be noted that the benefits of tree flattening grow as the syntactic complexity of designs increases. Thus, the improvement induced by tree flattening in the ATM example and the LOTOS examples is much bigger than in the other, less syntactically elaborate examples. Finally, caution should be used in interpreting the space-usage results, owing to the well-known difficulties in the space profiling of garbage-collected languages such as SML.

Table 1: Timings for PAC-generated Front Ends.

PM = "pattern matching"

+C = "pattern matching and caching"

+CF = "pattern matching, caching and tree flattening"

TIME (CPU seconds)					
System	Lang.	States	PM	+C	+CF
802-2	CCS	331	3.41	1.30	1.55
Mailer	CCS	1616	9.53	5.02	5.56
ATM	CCS	59614	723.54	189.44	83.56
Emitter	LOTOS	5571	361.35	58.69	55.06
Railway	LOTOS	19724	1244.61	363.05	171.74
Railway	PCCS	11905	2081.37	385.53	319.23

It should be noted that the original hand-written CWB semantic routines employed the same pattern-oriented approach to the calculation of semantic information, although neither call caching nor tree flattening were used.

Table 2: Space Usage for PAC-generated Front Ends.

PM = "pattern matching"

+C = "pattern matching and caching"

+CF = "pattern matching, caching and tree flattening"

SPACE (max process size in MB)					
System	Lang.	States	PM	+C	+CF
802-2	CCS	331	8.848	8.000	9.040
Mailer	CCS	1616	10.000	10.768	11.476
ATM	CCS	59614	54.846	42.144	54.756
Emitter	LOTOS	5571	15.256	14.932	15.128
Railway	LOTOS	19724	38.368	34.016	38.880
Railway	PCCS	11905	21.656	38.492	43.180

5 Conclusions and Directions for Future Work

This paper proposes a generic framework for the automatic verification of finite-state systems and shows how efficient tool support may be given for it. The framework consists of three basic concepts: (pre)bisimulations as a basis for refinement, the modal mu-calculus as a basis for model checking, and structural operational semantics as a basis for defining the semantics of design notations. The Concurrency Workbench of North Carolina and the Process Algebra Compiler of North Carolina exploit this framework to provide efficient yet easily customizable tool support based on these notions.

In the future we would like to investigate techniques for improving the space utilization of PAC-generated front ends. Recent work [3] also points to an abstract basis for model checking that circumvents the need for defining translations in the mu-calculus, and we would like to investigate the development of a model-checker generator based on these ideas. It could also be fruitful to look into the provision of generic support for symbolic approaches, such as those oriented around Binary Decision Diagrams [8]; steps in this direction may be found in [4]. Finally, it would also be interesting to investigate techniques for generically analyzing other kinds of systems, including those that pass values, exhibit real-time behavior, and have probabilistic aspects to their functioning.

References

- [1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87-152, 1992.
- [2] G. Bhat and R. Cleaveland. Efficient local model checking for fragments of the modal μ -calculus. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 107-126, Passau, Germany, March 1996. Springer-Verlag.

- [3] G. Bhat and R. Cleaveland. Efficient model checking via the equational μ -calculus. In *Eleventh Annual Symposium on Logic in Computer Science (LICS '96)*, pages 304–312, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
- [4] B. Bloom and A. Dsouza. Generating BDD models for process algebra terms. In P. Wolper, editor, *Computer Aided Verification (CAV '95)*, volume 939 of *Lecture Notes in Computer Science*, Liège, Belgium, June 1995. Springer-Verlag.
- [5] B. Bloom, S. Istrail, and A. Meyer. Bisimulation can't be traced. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages (POPL '88)*, pages 229–239, San Diego, January 1988. IEEE Computer Society Press.
- [6] G. Brebner. Private communication.
- [7] G. Bruns. *Distributed Systems Analysis with CCS*. Prentice-Hall, London, 1997.
- [8] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
- [9] CCITT. CCITT recommendation Z.100: Specification and description language SDL. Technical report, ITU General Secretariat, 1988.
- [10] U. Celikkan. *Semantic Preorders in the Automated Verification of Concurrent Systems*. PhD thesis, North Carolina State University, 1995.
- [11] U. Celikkan and R. Cleaveland. Generating diagnostic information for behavioral preorders. *Distributed Computing*, 9:61–75, 1995.
- [12] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
- [13] E.M. Clarke and J.M. Wing. Formal methods : state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.
- [14] R. Cleaveland, G. Luetzgen, V. Natarajan, and S. Sims. Modeling and verifying distributed systems using priorities: A case study. *Software Concepts and Tools*, 17:50–62, 1996.
- [15] R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In E. Brinksma, R. Cleaveland, K.G. Larsen, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 153–173, Aarhus, Denmark, May 1995. Springer-Verlag.
- [16] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

- [17] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag.
- [18] R. Cleaveland and S. Smolka. Strategic directions in concurrency research. *ACM Computing Surveys*, 28(4):607–625, December 1996.
- [19] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.
- [20] W. Elseaidy, R. Cleaveland, and J. Baugh. Modeling and verifying active structural control systems. *Science of Computer Programming*, To appear.
- [21] E.A. Emerson, C. Jutla, and A.P. Sistla. On model-checking for fragments of μ -calculus. In C. Courcoubetis, editor, *Computer Aided Verification (CAV '93)*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396, Elounda, Greece, June/July 1993. Springer-Verlag.
- [22] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Symposium on Logic in Computer Science (LICS '86)*, pages 267–278, Cambridge, Massachusetts, June 1986. IEEE Computer Society Press.
- [23] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1989/1990.
- [24] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [25] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.
- [26] D. Kozen. Results on the propositional μ -calculus. *Theoretical Computer Science*, 27:333–354, 1983.
- [27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Berlin, 1992.
- [28] R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.
- [29] International Standards Organization. LOTOS – a formal description technique based on the temporal ordering of observational behavior. Technical Report ISO/TC 97/SC 21 N 1573, ISO, February 1987.
- [30] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.
- [31] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

Automatic Concurrency in SequenceL

By

Daniel E. Cooke
Department of Computer Science
Texas Tech University
P.O. Box 43104
Lubbock, TX 79409
dcooke@coe.ttu.edu

Vladik Kreinovich
Department of Computer Science
University of Texas at El Paso
500 West University
El Paso, TX 79968
vladik@cs.utep.edu

Abstract.

This paper presents a candidate language solution for the automation of data parallel solutions of concern to the oil industry and U.S. Federal Agencies involved in the analysis of Satellite Telemetry Data. Focus is placed upon major language issues facing the development of the information power grid. The paper presents an example of the type of parallelism desired in the Grid and gives a JAVA solution. The JAVA solution contains artifacts of the design of parallel solutions. The same problem is then recast in the high level language *SequenceL*, in which parallelisms are implied. The *SequenceL* approach seems to be a good candidate for a Grid Oriented Language, in that the abstraction relieves the problem solver of much of the burden normally required in development of parallel problem solutions.

The Need for New Language Abstractions

Hardware improvements and the general spread of computing and computer applications have created opportunities for scientists and engineers to solve ever more complicated problems. However, there are concerns about whether scientists and engineers possess the software tools necessary to solve these problems and what computer scientists can do to help the situation.

The fundamental software tool for problem solving is the programming language. A programming language provides the abstraction employed in solving problems. In order to keep pace with hardware improvements, computer scientists should continually address the problem of language abstraction improvement. When advances in hardware make problems technically feasible to solve, there should be corresponding language abstraction improvements to make problems *humanly feasible* to solve.

In the recent past, most language studies have resulted in the addition of new features to existing language abstractions. The most significant changes have resulted in additions to language facilities for the definition of program and data structures. These changes have primarily taken place to accommodate the needs for concurrent execution and software reuse. Although it is important to add to the existing abstractions to satisfy immediate technical problems, research also needs to be undertaken to simplify and minimize existing abstractions.

There are application domains where the need for simpler language abstractions is of vital importance. There are estimates that less than 1% of the available satellite data has been analyzed. [4] There exists the ability to acquire and store the data, but weakness in the ability to determine its information content. Soon NASA will have satellites in place that, in sum, will produce a terabyte of data per day. A major problem associated with the analysis of the data sets is the time needed to write the medium-to-small programs to explore the data for segments containing information pertinent to particular earth science problems. Software productivity gains in developing exploratory programs are needed in order to enhance the abilities of earth scientists in their efforts to grapple with the complexity and enormity of satellite and seismic data sets. Software productivity gains can be accrued through languages developed out of foundational research focusing on language design.

The need for computer language abstraction improvement is even more pronounced given the desire to develop distributed approaches to data analysis. Currently, industry and government agencies are paying a lot of attention to approaches involving complicated data parallel solutions. Data parallelisms embody the idea of "scatter/gather" approaches to problem solving. The basic idea is to have a Single-Instruction-Multiple-Data (SIMD-) type architecture where a single

program executes on multiple, networked processors. Each processor analyzes a piece of a large data set (i.e., the data set is scattered among several processors) and when processing on the partial data sets is completed, the partial solutions are assembled (i.e., gathered) to produce a single large solution set. This "scatter/gather" approach to computing has been very successful in the analysis of seismic data sets.

Prior to the SIMD approach, the oil industry would analyze entire data sets on a single "super computer." The SIMD's and has since resulted in cheaper and faster processing of seismic data sets. These data sets are used to determine which sites companies should lease for their offshore drilling activities. The seismic data sets (upon which scatter/gather approaches have proven to be successful) have quite a bit in common with the satellite telemetry data sets that NASA and other federal agencies acquire and store. There is a major effort to generalize the SIMD architecture by developing a super system that could employ idle resources on the World Wide Web. The effort is generally called the Information Power Grid or the Grid for short.

Background on The Information Power Grid.

The Information Power Grid is a major effort being funded by a number of U.S. federal agencies including NASA and the NSF. The goal of this effort is to establish a computing infrastructure on the world wide web, providing powerful supercomputing level resources to any user connected to the web. "The grid will connect multiple regional and national computational grids to create a universal source of computing power. The word "grid" is chosen by analogy to the electric power grid, which provides pervasive access to power..." [8].

One way to envision the goal of this effort is to imagine a web browser button that would allow the user to submit programs for execution. In an ideal case, the program would be analyzed to determine the parallelisms it contains. Then, a suitable distributed, parallel architecture would be configured by seizing idle processors connected to the internet - the envisioned system would provide to all entities connected to the web, access to teraflop computing capabilities.

Clearly there are a number of technical challenges that face those who are developing the grid. The focus here is on the computer language issues.

Powerful new strategies for supporting the development of high-performance distributed applications will be needed.... The application developer should be able to concentrate on problem analysis and decomposition at a fairly high level of abstraction... To do this, [the programming support system] will need to find every possible type of parallelism within the application, including data parallelism and task or object parallelism... From the user's perspective, the most appealing approach to program decomposition is automatic parallelism. [9]

In this paper, we will focus on language solutions to the programming support system referred to in the preceding passage. We will first show a simple data parallel problem solution using JAVA's multithreading features. We will then describe the very high level language, *SequenceL*, and indicate how the same data parallel problem solution is easily identifiable in the *SequenceL* solutions. One goal of the paper is to convince the reader that *SequenceL* holds promise as a grid-oriented language.

Data Parallelisms in JAVA.

The key to achieving high performance on distributed-memory machines is to allocate data to various processor memories to maximize locality and minimize communication. Data parallelism is parallelism that derives from subdividing the data domain in some manner and assigning the subdomains to different processors. [9] Data parallelisms (i.e., those characteristic of SIMD-type architectures) typically result in the same computation being performed simultaneously on subdivided data sets, as opposed to dividing up the computation itself.

Consider the following word-search problem as written for sequential execution in JAVA:

```
String s = "here is a test string";
String s1 = "test";

char[] sample = s.toCharArray();
char[] find = s1.toCharArray();

System.out.println(sample);

n = sample.length;
n1 = find.length;

for(i=0;i<=n-n1;i++)
    {System.out.println(s.substring(i,i+n1));
      if(s.substring(i,i+n1).equals(s1))
        {System.out.println(" TRUE == FOUND ");}
    }
```

The sample text is a 21 character string. In this problem, the goal is to determine if the 4 character string "test" is in the sample. The linear search involves checking each unique 4 character substring of the sample, to see if it is equal to the string "test".
if-statement is executed 18 times.

A data parallel solution to this problem involves the separation of the 18 unique 4 character substrings. This is accomplished in Exhibit 1. In Exhibit 1 an array of reference variables is declared (in line 33). The array is based upon the difference in length of the string being searched and the length of the string for which the search is being conducted (i.e., in this example, the array will consist of 18 elements). In lines 35-38, the string being searched is subdivided and used to initialize the instance variables as the 18 references to the class are instantiated via the class' constructor method *wrdsrch2* (lines 7-12). Once the 18 instances are set up, the concurrent processes to compare the strings are initiated (in lines 42-43). The 18 comparisons are executed concurrently. Execution of the *main* method proceeds no further until the 18 processes are joined (in lines 45-48). The 18 instances of the boolean variable *found* are then printed as output.

The concurrent solution to this problem is not easily found (when one studies the sequential version) and the concurrent solution is difficult to design, write, and understand. Furthermore, the concurrent solution exhibits artifacts of the design effort to produce the concurrent solution. These artifacts are the *thread's*, *try's*, *join's*, and *run()*. The next four sections of the paper are intended to convince the reader that the high level, executable language, *SequenceL* may provide a more suitable abstraction for representing data parallelisms.

Introducing the SequenceL Language Constructs.

SequenceL was introduced as an approach to software development that offers a different, and for many, a more intuitive approach to problem solving. [2, 3, 4, 5, 6, 7] The assumption underlying the design of *SequenceL* is that the data product, as produced by software, is the true product of the software developer.

Using traditional languages, programmers write explicit algorithms that imply data products. The goal of the *SequenceL* design effort is to provide a language in which specifiers make an explicit statement of the data product, which in turn implies the algorithm. Whereas algorithm-writers must come to know and understand the implied data product, a data-product specifier need not know the implied algorithm. In *SequenceL*, focus turns from the matter that produces the product to the product itself.

One of the main difficulties in traditional programming is grasping the true nature of the implied data product. Implied items are elusive and often require a large amount of concentration to fully grasp. The effort to gain the understanding of the data product impedes productivity. Complex data products are typically recursively or iteratively defined. Software engineers have long realized that the construction of loops is complex and costly. [10] Bishop noted that "Since Pratt's paper on the design of loop control structures was published more than a decade ago, there has been continued interest in the need to provide better language features for iteration." [1]

SequenceL possesses no iterative constructs and accommodates a unique form of recursion where functions may embed themselves among intermediate data results. *SequenceL* is a language for describing a data product in terms of both form and content. The difference between the traditional approach to programming and the *SequenceL* approach is precisely the difference between an implicit product and an explicit statement of the product. Consider as an example a simple program to compute the *mean* value of an unknown number of data values. For example, if the values are (10,25,30,35,40), then the *mean* is obtained by:

$$\text{Mean} = (10 + 25 + 30 + 35 + 40) \div 5$$

In the traditional approach one states an algorithm (i.e., a step-by-step sequence of instructions) that will produce the desired result. In *SequenceL*, one declares the desired data product:

Traditional Approach - Pseudo Code

1. Read in the numbers, one at a time, counting them as they are read.
2. Add the values together (Sum them).
3. Divide the sum by the count obtained in step (1).

SequenceL Approach - Pseudo Code

Divide the sum of the values by the number of values.

SequenceL consists of three constructs that can be combined in any manner to declare a data product. All *SequenceL* operators operate on and produce only sequences. Sequences can be scalar (i.e., singleton), nonscalar (i.e., lists of singletons), and nested structures (i.e., lists of lists). Nested structures can be nested to any depth.

The first *SequenceL* construct is realized in the definition of the built-in arithmetic and relational comparison operators and is called the *regular* construct. The regular construct applies an operator to all elements of the operand sequences, be they singletons, nonscalars, or nested sequences:

Two Singletons: $+$

4
4

 $=$ [8]

Nonscalars: $+$

4
4
3
2

 $=$ [13]

Nested: $+$

10	4
20	5
30	6
40	7
50	8

 $=$

14
25
36
47
58

In all cases these operations execute in a uniform way: *the operator is applied to corresponding elements of the normalized operand sequences*. In the first two examples, the operator applies to corresponding elements of singleton sequences: the first example having a binary set of singletons and the second example having 4 singleton sequences. The last (i.e., the nested) example applies the operator to corresponding elements of nonsingleton sequences.

In the examples above normalization has no affect. Normalization affects nested operands that are of varying levels of nesting and/or of varying cardinalities. The operands are normalized in terms of size, prior to carrying out the assigned operation. In terms of cardinality or nesting differences, the elements of the smaller operands are repeated in the order they occur until the smaller operand is equal in size to the largest operand:

$+$

10	4
20	5
30	6
40	
50	

 $=$ *normalize* $+$

10	4
20	5
30	6
40	4
50	5

 $=$

14
25
36
44
55

The *generative* construct allows for the expansion of sequences. The simple form expands integers within some bounds. The nontrivial version expands values within bounds when the values satisfy a constraining formula:

Simple

1
:
5

 $=$

1
2
3
5

 Nontrivial:

0
1
:
+(p(p(next)),p(next))
:
8

 $=$

0
1
1
2
3
5
8

The *irregular* construct applies an operator selectively. The selection may be based upon the content or the form of operand sequences. Often a *when-clause* is employed in order to accomplish the selection. When the condition of the when clause is true, the operator is applied:

$(*(salary(i),1,1) \quad \text{when} \quad evaluation(i) > 5 \quad)$

The operation will apply to each *i*th salary, when the corresponding *i*th evaluation is better than 5.

Selection can also be based upon the form of operand sequences. In these cases, a *Using-clause* accompanies the function. Consider the *SequenceL* solution for the matrix multiply:

Mmultiply(Consume(pred(n1,n2),succ(m1,m2)), Produce(next)) where next(x,y) =

+ (* (pred(x,all), succ(all,y)))

Using x ,y From [1,...n1] * [1,...,m2]

In this solution, subscripts x and y obtain their values from an ordered, cartesian-like product obtained from the sequences of values from 1 to $n1$, inclusively and from 1 to $m2$, inclusively. The generative construct is employed to produce the desired sequences. The resulting x,y -pairs are [<1,1>, <1,2>, ..., <2,m2>, ..., <n1,1>, <n1,2>, ..., <n1,m2>].

In addition to the Cartesian operator (*) employed in the *Using-clause* of the example above, subscript sequences can be combined with an intersection (&), union (OR), or difference (\) operator. The example above also introduces the *wild-card* subscript *all*, which obtains all values of the selected sequence dimension.

The individual (x,y) values of the resulting product matrix are computed as regular computations: summing the sequences of products obtained by multiplying rows of a predecessor matrix by corresponding columns of a successor matrix.

SequenceL functions are built-up using the regular, irregular, and generative constructs. These constructs can be combined in any manner and the sequences they produce can be used as operands or subscripts to operands. Sequences and functions can both serve as the result of a function's execution. The final data product will consist solely of sequences, while intermediate results can be a combination of sequences and functions.

The functions themselves execute in a data dependent fashion based upon their signatures. Domain operands are consumed from the database to which a function is applied. The next section describes a computational model that embodies a simplified version of *SequenceL*'s execution strategy.

SequenceL Computational Model.

SequenceL's execution strategy is based upon a data dependency approach to execution. A simple form of the computational model is presented in this section. First, we introduce the following sets and mappings:

Set Definitions and Mappings:

S	= set of all possible sequences
N	= set of all possible function symbols
D	= $2^{\{pred,succ\}}$
F	= $S \cup (S \times N \times S) \cup (S \times N) \cup (N \times S)$
Π	= 2^F
P	$\in \Pi$
H:	$N \rightarrow D$
B:	$F \times S \rightarrow F \cup \{undefined\}$

The sets S and N are the sets of all possible sequences and all possible function symbols (i.e., *SequenceL* function names), respectively. Set D is the set of all possible function domain arguments, expressed as the power set of the set {pred, succ}. The set Π is the power set of all possible strings containing:

1. A sequence;
2. a sequence, followed by a function name, followed by a sequence;
3. a sequence, followed by a function name; or
4. a function name, followed by a sequence.

A program P is an element of Π . The mappings of the computational model are as follows:

1. H: which maps from a function symbol to its matching domain arguments; and
2. B: which maps from a function symbol and a sequence or sequence-pair, to an element of F.

Next we have a set defined based upon a given program P . A function of P is enabled for execution if all domain arguments are available. Given a generalized string containing a (possibly empty) sequence α , a function symbol f , and a (possibly empty) sequence β , the function is enabled for execution if two conditional statements are true. The first conditional statement requires α to be a nonempty string if f 's domain set contains *pred*. Likewise, the second conditional statement requires β to be a nonempty string if f 's domain set contains *succ*:

$$\text{Enabled}(P) = \{ \alpha f \beta \mid \alpha f \beta \subseteq P \ \& \ \\ (H(f) \supseteq \{\text{pred}\} \Rightarrow \alpha \neq \lambda \ \& \ \alpha \in S) \ \& \ \\ (H(f) \supseteq \{\text{succ}\} \Rightarrow \beta \neq \lambda \ \& \ \beta \in S) \ \& \ \\ B(\alpha f \beta) \neq \text{undefined} \};$$

Where λ is the empty string, $\Rightarrow$ is logical implication, and $\subseteq_s$ is a substring operator.

A computation replaces all enabled functions and their surrounding sequences from the program P with the string of sequences and/or function symbols computed by an enabled function. For any i , γ_i and δ_i are possibly empty strings. The string computed by an enabled function is determined by the function symbol and its arguments α and β . If there are no enabled functions, P is obtained. The execution is meant to allow for concurrent processing of all enabled functions when: $|\text{Enabled}(P)| > 1$.

$$\text{Execute}(P) = \{ \gamma_1 B(\delta_1) \gamma_2 B(\delta_2) \dots \gamma_n B(\delta_n) \gamma_{n+1} \mid P = \gamma_1 \delta_1 \gamma_2 \delta_2 \dots \gamma_n \delta_n \gamma_{n+1} \ \& \ \delta_1, \delta_2, \dots, \delta_n \in \text{Enabled}(P) \}$$

$$\text{if } \text{Enabled}(P) \neq \emptyset$$

$$\text{Execute}(P) = P \quad \text{if } \text{Enabled}(P) = \emptyset$$

A total (proper) computation is defined as a sequence $P_1, P_2, \dots, P_n$, where P_1 is the initial program and $P_{i+1} \in \text{Execute}(P_i)$, and $\text{Enabled}(P_n) = \emptyset$.

Now consider an example of the execution of a *SequenceL* operator:

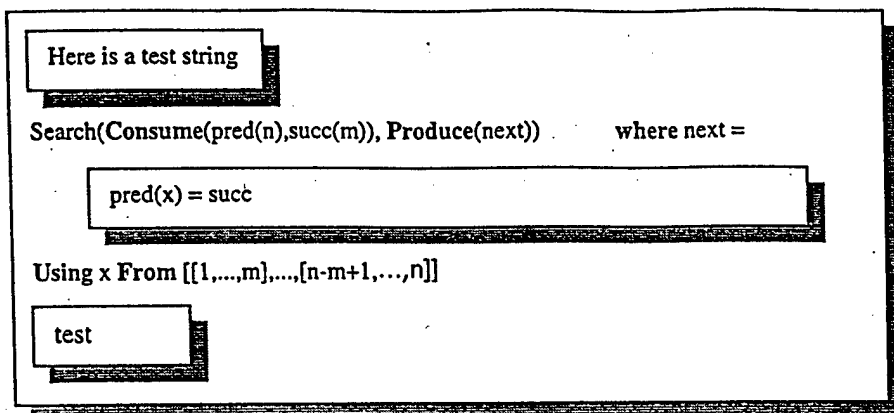
$$\begin{aligned} P &= (4+5)/(5-2) \\ \text{Enabled}(P) &= \{4+5, 5-2\} \\ \text{Execute}(P) &= \{P', P'', P'''\} \text{ where } \begin{aligned} P' &= (B(4+5))/(5-2) = (9)/(5-2) \\ P'' &= (4+5)/(B(5-2)) = (4+5)/(3) \\ P''' &= (B(4+5))/(B(5-2)) = (9)/(3) \end{aligned} \end{aligned}$$

Notice that P''' provides for the concurrent solution so that

$$P_1 = (4+5)/(5-2), P_2 = (9)/(3), P_3 = (3)$$

Data Parallelisms in SequenceL.

Data parallelisms in *SequenceL* are definable in a straightforward manner through the use of the data dependent execution strategy of *SequenceL* functions. Given the following database configuration, the *word-search* example is represented in an intuitive manner:



In this example, the cardinalities of the predecessor and the successor are obtained in identifiers n and m , respectively. Based upon the *using clause*, the *predecessor's* subscript x obtains values (in order) from the generated sequences:

```
[[1,2,3,4],[2,3,4,5],[3,4,5,6],[4,5,6,7],[5,6,7,8],[6,7,8,9],[7,8,9,10],[8,9,10,11],[9,10,11,12],
[10,11,12,13],[11,12,13,14],[12,13,14,15],[13,14,15,16],[14,15,16,17],[15,16,17,18],
[16,17,18,19],[17,18,19,20],[18,19,20,21]]
```

The *using clause* helps subdivide the larger data set into 18 smaller sets – much like the parceling of data accomplished in lines 35-39 and 7-12 in the JAVA version presented in Exhibit 1. The function results in the following set of relations being added to the *SequenceL* program:

```
[[here] = [test], [ere] = [test], [re i] = [test], [e is] = [test], [is] = [test], [is a] = [test],
[s a] = [test], [a i] = [test], [a te] = [test], [tes] = [test], [test] = [test], [est] = [test],
[st s] = [test], [t st] = [test], [str] = [test], [stri] = [test], [trin] = [test], [ring] = [test]]
```

The concurrent evaluation of the resulting conditions is now clearly implied due to the computational model of *SequenceL* - a model that allows the execution of any function or operator as soon as the data required for the operator or function is available:

```
[[here] = [test] || [ere] = [test] || [re i] = [test] || [e is] = [test] || [is] = [test] || [is a] = [test] ||
[s a] = [test] || [a i] = [test] || [a te] = [test] || [tes] = [test] || [test] = [test] || [est] = [test] ||
[st s] = [test] || [t st] = [test] || [str] = [test] || [stri] = [test] || [trin] = [test] || [ring] = [test]]
```

After concurrent evaluation, the vector of boolean results remains in the database:

```
[ false, false, false, false, false, false, false, false, false, false, true, false, false, false, false, false, false, false ]
```

The parallelisms in *SequenceL* are more intuitive in that the parallelisms do not result in the separation of elements of functionality and, since parallelisms are implied, the solution does not require the use of additional constructs as is seen in the *thread*, *run()*, *try*, etc. required in the JAVA concurrent solution.

Summary.

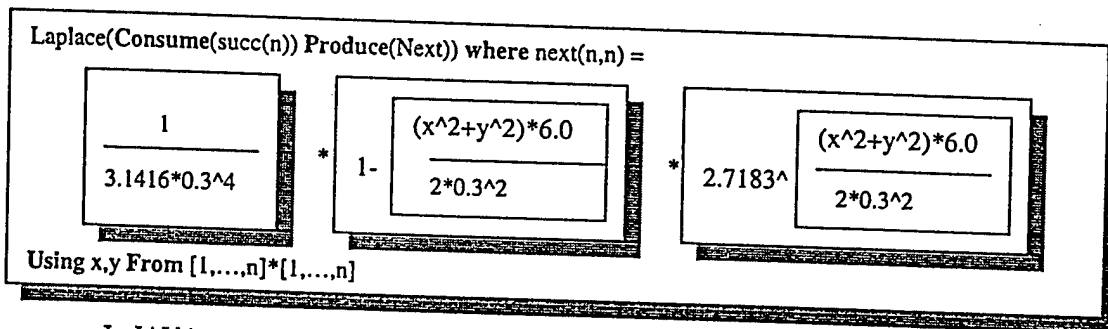
SequenceL seems to provide a more intuitive approach to data analysis problems – especially when parallelisms are required in the solution. Even the most modern computing languages (e.g., JAVA) are cumbersome when it comes to the design and understanding of parallel solutions. Modern approaches to data analysis as exemplified by the goals of the Grid project require languages that can express parallelisms at a higher level - languages for which parallelisms can be identified automatically.

SequenceL, a fully computable language, is presented as a candidate Grid Oriented Language - a language for expressing the complicated parallelisms that will dominate Grid applications. *SequenceL* is a good candidate because one can express problem solutions at a high level and because there is little the programmer must do in the way of explicitly defining parallelisms. Task and Vector parallelisms are implied by the *SequenceL* evaluation of nested terms and regular expressions, respectively. Data parallelisms are realized through an interaction of the irregular and generative constructs, which provide

for the parceling of data, and through the concurrent, data-dependent execution strategy followed in the evaluation of functions and operators. Although the example data parallel problem solution developed in this paper is rather simple, the example scales up to many real-world data mining problems involving image processing and security-based text searches. The searching of image databases follows the same parceling and scatter/gather approach to programming. The difference is that the objects for which one is searching is characterized mathematically. These mathematically defined objects serve as a *kernel*, which drives the search much like the string *test* served as the object for which the text string search was done in this paper. An example of an image-based kernel is the following Gaussian-Laplacian operator often employed for edge detection:

$$GL(x)(y) = \frac{1}{\pi 0.3^4} \left[1 - \frac{(x^2 + y^2) 6.0}{2 * 0.3^2} \right] 2.7183^{\left[\frac{(x^2 + y^2) 6.0}{2 * 0.3^2} \right]}$$

The operator is applied for values from 1 to *n* for *x* and *y*. The *SequenceL* function corresponds directly to the mathematical formula:



In JAVA, as in other languages, the *thread's*, *try* 's, and *run()*'s are artifacts of a design effort to parallelize a problem solution. There are no such artifacts in the *SequenceL* solutions largely because the problem solver is relieved of much of the design effort due to the unique constructs of this high level language.

```

1  class wrdsrch2 extends Thread{
2  String text;
3  String target;
4  boolean found;
5  int i;
6
7  wrdsrch2(String in, String targ, int k)    {
8      target=targ;
9      text=in;
10     found=false;
11     i=k;
12 }
13
14 public void run()    {
15     if(text.equals(target))
16         {found = true;}
17 }
18
19 public static void main (String args[]) {
20     int i, j, k, n, n1;
21
22     String s = "here is a test string";
23     String s1 = "test";
24     char[] sample = s.toCharArray();
25     char[] find = s1.toCharArray();
26
27     System.out.println(sample);
28
29     n = sample.length;
30     n1 = find.length;
31     String send;
32
33     wrdsrch2 w[] = new wrdsrch2[(n-n1)+1];
34
35     for(i=0;i<=n-n1;i++)
36         {send = s.substring(i,i+n1);
37          w[i] = new wrdsrch2(send,s1,i);
38          }
39
40     System.out.println("To Run ");
41
42     for(i=0;i<=n-n1;i++)
43         {w[i].start();}
44
45     for(i=0;i<=n-n1;i++)
46         {try {w[i].join();
47          catch (InterruptedException ignored) { }
48          }
49
50     System.out.println("The answer is: ");
51
52     for(i=0;i<=n-n1;i++)
53         {System.out.println(w[i].found);}
54
55 }

```

Exhibit 1. Data Parallelism in a Word Search Problem.

References.

- [1] J. Bishop, "The Effect of Data Abstraction on Loop Programming Techniques," *IEEE Trans. Soft. Eng.* Vol. SE-16, Number 4, April 1990, pp. 389-402.
- [2] D. E. Cooke, "Possible Effects of the Next Generation Programming Language on the Software Process Model," *International Journal on Software Engineering and Knowledge Engineering*, Vol. 3, No. 3, September 1993, pp. 383-399.
- [3] D. Cooke, E. Demirors, O. Demirors, A. Gates, B. Kraemer, and M. M. Tanik, "Languages for the Specification of Software," *Journal of Systems and Software*, 1996, pp. 269-308.
- [4] "An Introduction to SEQUENCEL: A Language to Experiment with Nonscalar Constructs," *Software Practice and Experience*, Vol. 26, No. 11, November 1996, 1205-1246.
- [5] D. Cooke, "SequenceL Provides a Different way to View Programming," *Computer Languages* 24 (1998) 1-32.
- [6] D. E. Cooke, "The Semantics of SequenceL", to appear in *Journal of Programming Languages*.
- [7] D.E. Cooke and J.E. Urban, "The Application of the SequenceL Language to Complicated Database Applications," to appear in the *Proceedings of the Application-Specific Software Engineering and Technology Workshop*, Dallas, Texas, March 27-28, 1998.
- [8] Ian Foster and Carl Kesselman, Preface to *The Grid, Blueprint for a New Computing Infrastructure*, (Ian Foster and Carl Kesselman, editors) Morgan Kaufmann Publishers, San Francisco, CA (1999).
- [9] Ken Kennedy, "Compilers, Languages, and Libraries," in *The Grid, Blueprint for a New Computing Infrastructure*, (Ian Foster and Carl Kesselman, editors) Morgan Kaufmann Publishers, San Francisco, CA (1999).
- [10] H. Mills and R. Linger, "Data Structured Programming: Programming without Arrays and Pointers," *IEEE Trans. Soft. Eng.* Vol. SE-12, Number 2, February 1986, pp. 192-197.

Acknowledgement

Research sponsored under NASA NCCW-0089.

On Methodology of Representing Knowledge in Dynamic Domains

Michael Gelfond and Richard Watson

Computer Science Department

University of Texas at El Paso

El Paso, Texas 79968

(915) 747-6957, (915) 747-5596

{mgelfond,rwatson}@cs.utep.edu

ABSTRACT

The main goal of this paper is to outline a methodology of programming in dynamic problem domains. The methodology is based on recent developments in theories of reasoning about action and change and in logic programming. The basic ideas of the approach are illustrated by discussion of the design of a program which verifies plans to control the Reactive Control System (RCS) of the Space Shuttle. We start with formalization of the RCS domain in an action description language. The resulting formalization $\mathcal{A}_{RCS}$ together with a candidate plan α and a goal G are given as an input to a logic program. This program verifies if G would be true after executing α in the current situation. A high degree of trust in the program's correctness was achieved by

(a) the simplicity and transparency of our formalization, $\mathcal{A}_{RCS}$, which made it possible for the users to informally verify its correctness;

(b) a proof of correctness of the program with respect to $\mathcal{A}_{RCS}$.

This is an ongoing work under a contract with the United Space Alliance - the company primarily responsible for operating the Space Shuttle.

{Keywords: Action Languages, Logic Programming, Agents}

1 INTRODUCTION

The main goal of this paper is to outline a methodology of programming in dynamic problem domains, based on recent developments in theories of actions and change [14], [13], and [8]. These theories provide a basis for reasoning about worlds inhabited by intelligent agents, i.e., by entities that have goals they want to achieve, actions they can perform, and knowledge of the effects of these actions and of the surrounding environment. To perform nontrivial reasoning an intelligent agent situated in a changing domain needs the knowledge of causal laws that describe effects of actions that change the domain, and the ability to observe and record occurrences of these actions and the truth values of fluents¹ at particular moments of time. One of the central problems of knowledge representation is the discovery of methods of representing this kind of information in a form allowing various types of reasoning about the dynamic world and at the same time tolerant to future updates. Our description of dynamic domains will be based on the formalism of action languages. Such languages, first introduced in [11], can be thought of as formal models of the part of the natural language that are used for describing the behavior of dynamic domains. An action language can be represented as the combination of two distinct parts: an "action description language" and an "action query language". A set $\mathcal{A}$ of propositions in an action description language, called an *action description*, describes the effects of actions on states. Mathematically, it defines a transition system with nodes corresponding to possible states and arcs labeled by actions from the given domain. An arc (σ_1, a, σ_2) indicates that an execution of action a in state σ_1 may result in the domain moving to the state σ_2 . An action query language serves for expressing properties of paths² of a given transition system. The syntax of such a language is defined by two classes of syntactic expressions: *axioms* and *queries*. The semantics of the action language is defined by specifying, for every action description $\mathcal{A}$, every set Γ of axioms, and every query Q , whether Q is a consequence of Γ in $\mathcal{A}$ ($\Gamma \models_{\mathcal{A}} Q$). This relation is in general *non-monotonic*, i.e. addition of new information to $\mathcal{A}$ and/or Γ can force a reasoner to

¹In this paper by fluents we mean (time-dependent) properties of objects of a dynamic domain.

²By a path of a transition system T we mean a sequence $\sigma_0, a_1, \sigma_1, \dots, a_n, \sigma_n$ such that for any $1 \leq i < n$, $\sigma_i, a_{i+1}, \sigma_{i+1}$ is an arc of T . σ_0 and σ_n are called initial and final states of the path respectively.

withdraw its previous conclusion about Q . Action theories can be used by system designers to specify domains in which agents are expected to act and the desired behavior of the agents. Such specifications allow designers to reason about agents behavior and verify its correctness. Action descriptions and axioms can also be used to supply an agent with the knowledge about its domain and its abilities to act. In this case this knowledge can be used by the agent to assimilate observations, select goals, plan (or replan) to achieve the selected goal, and act accordingly. Action theories also play an important role in high-level robot control languages. A typical command of such language, say, "if $\Gamma \models_A Q$ then execute action a " refers explicitly to action description A and axioms Γ containing current knowledge of the robot and to the consequence relation of the corresponding action theory. In this paper the use of action description languages will be illustrated by their application for modeling subsystems of the Space Shuttle. This is an ongoing work under a contract with United Space Alliance (USA) - the company primarily responsible for operating the Space Shuttle. For our initial research we selected the Reaction Control System (RCS) of the Space Shuttle. An action description of the RCS was created and tested. The resulting query answering system was used to allow flight controllers to automatically verify plans for operation of the RCS. Our goal was to create a system with several important characteristics. First it had to be usable by people without much training in Computer Science, and be easily modifiable and adaptable to modeling other subsystems of the Shuttle. This was achieved by introducing users to the syntax and informal semantics of action description languages and by hiding all other details of implementation. Second, we wanted to have a very high degree of trust in the systems's correctness. Partly it was achieved by the simplicity and transparency of our description of the RCS which made it possible for people from the USA to informally verify correctness of our representation. The corresponding plan checking program was written in a logic programming language and its correctness with respect to our representation was proven mathematically. This proof was developed in conjunction with writing a program and relied heavily on recent advances in logic programming [12], [3], [1], and [2]. The program was implemented by gradual transformation of the initial specification into an executable program. The proof insured correctness of these transformations. In the next section we give a short introduction into a syntax and semantics of an action description language $\mathcal{L}_0$ used for modeling the RCS and give examples of its use. The description of the RCS will consists of an action description A containing description of effects of actions which can be performed by flight controllers, and the collection Γ of axioms describing the current state of the system. The plan checking task can be reduced to verifying that $\Gamma \models_A \text{holds_after}(\text{goal}, \text{plan})$ where $\text{holds_after}(g, \alpha)$ says that the sequence α of actions is executable and that the goal g would be true if α were executed. The remaining sections will contain a short introduction to action languages, logic programming, the description of the logic program computing the consequence relation $\models_A$, and the corresponding correctness theorems.

2 LANGUAGE $\mathcal{L}_0$

In this section we define an action language $\mathcal{L}_0$ which can be viewed as the combination of action description language $\mathcal{B}_0$ and query description language $\mathcal{Q}_0$. We assume a fixed signature Σ_0 which consists of two disjoint, nonempty sets of symbols: the set F of fluents and the set A of actions. Signatures of this kind will be called *action signatures*. By fluent literals we mean fluents and their negations. Negation of $f \in F$ will be denoted by $\neg f$. Fluent literals f and $\neg f$ are called *contrary*. By $\bar{l}$ we denote the fluent literal contrary to l . A set $S \subseteq F$ is called complete if for any $f \in F$ $f \in S$ or $\neg f \in S$.

Action description language $\mathcal{B}_0$ provides a simple and elaboration tolerant way to describe transition systems. The action descriptions of $\mathcal{B}_0$ consist of arbitrary collections of propositions of the form

$$\text{causes}(a, l_0, [l_1, \dots, l_n]) \quad (1)$$

$$\text{causes}([l_1, \dots, l_n], l_0) \quad (2)$$

$$\text{impossible}(a, [l_1, \dots, l_n]) \quad (3)$$

where a is an action and $l_0, \dots, l_n$ are fluent literals. In each of the propositions above, l_0 is called the *head* of the proposition and $[l_1, \dots, l_n]$ is called the *body* of the proposition. A proposition of the type 1 says that, if the action a were to be executed in a situation in which $l_1, \dots, l_n$ hold, the fluent literal l_0 will be caused to hold in the resulting situation. Such propositions are called *dynamic causal laws*. A proposition of the type 2, called a *static causal law*, says that the truth of fluent literals, $l_1, \dots, l_n$, in an arbitrary situation, s , is sufficient to cause the truth of l_0 in that

situation. A proposition of the type 3 says that action a can not be performed in any situation in which $l_1, \dots, l_n$ hold.

In addition to the propositions above, we allow *definition propositions* which will be viewed as a shorthand for specific sets of static causal laws. Definition propositions have the form:

$$\text{definition}(l_0, [l_1, \dots, l_n]) \quad (4)$$

where $l_0, \dots, l_n$ are fluent literals. The following restrictions apply to the use of such propositions:

1. Neither l_0 nor $\bar{l}_0$ belong to the head of any of the static or dynamic causal laws.
2. There are no definitions whose heads are contrary fluent literals.

Let $\{\text{definition}(l_0, \beta_1), \dots, \text{definition}(l_0, \beta_n)\}$ be the set of all definitions, in an action description $\mathcal{A}$, which contain l_0 in the head. The fluent literal l_0 is true in any situation in which at least one of β 's is true. Otherwise it is false. As was mentioned, definition propositions are a shorthand for a larger set of static causal laws. The above definitions of l_0 can be replaced by static causal laws as follows:

1. For each proposition, $\text{definition}(l, \beta_i)$, add a static causal law $\text{causes}(\beta, l)$.
2. For each set of literals, θ , such that:
 - (a) θ is consistent,
 - (b) for each β_i there exists a literal $l \in \beta_i$ such that $\bar{l} \in \theta$, and
 - (c) there is no subset of θ which satisfies conditions (a) and (b),
add a static causal law $\text{causes}(\theta, \bar{l})$.

An action description $\mathcal{A}$ of $\mathcal{B}_0$ defines a transition system describing effects of actions on the possible states of the domain. By a state we mean a consistent set σ of fluent literals such that

1. σ is complete;
2. σ is closed under the static causal laws of $\mathcal{A}$, i.e. for any static causal law (2) of $\mathcal{A}$, if $\{l_1, \dots, l_n\} \subseteq \sigma$ then $l_0 \in \sigma$.

States serve as the nodes of the transition diagram. Nodes σ_1 and σ_2 are connected by a directed arc labeled by an action a if σ_2 may result from executing a in σ_1 . The set of all states that may result from doing a in a state σ will be denoted by $\text{res}(a, \sigma)$. Precisely defining this set for increasingly complex action descriptions seems to be one of the main difficulties in the development of action theories. In case of action descriptions from $\mathcal{B}_0$ we will use the approach suggested in [18].

We will need the following auxiliary definitions. We say that an action, a , is *prohibited* in a state, σ , if $\mathcal{A}$ contains a statement $\text{impossible}(a, [l_1, \dots, l_n])$ such that $[l_1, \dots, l_n] \subseteq \sigma$. Let F be a set of fluent literals of $\mathcal{A}$. By the *causal closure* of F we mean the least superset, $Cn_R(F)$, of F closed under the static causal laws of $\mathcal{A}$. By $E(a, \sigma)$ we denote the set of all fluent literals, l_0 , for which there is a dynamic causal law $\text{causes}(a, l_0, [l_1, \dots, l_n])$ in $\mathcal{A}$ such that $[l_1, \dots, l_n] \subseteq \sigma$. We say that a state σ' may result from doing action a in a state σ if

1. a is not prohibited in σ ;
2. σ' satisfies the condition $\sigma' = Cn_R((\sigma \cap \sigma') \cup E(a, \sigma))$

An action description is called *deterministic* if for any action, a , and state, σ , there is at most one state, σ' , satisfying that above conditions. An action description is called *consistent* if $\text{res}(a, \sigma) = \emptyset$ iff a is prohibited in σ .

One may observe that the complete understanding of the formal semantics of $\mathcal{B}_0$ requires some effort. Fortunately this effort is not necessary for most users of the language. Similar to other programming and specification languages the complete understanding is needed only if one wants to prove correctness of compilers and/or various properties of programs of $\mathcal{B}_0$. Otherwise, informal understanding of the meaning of propositions of $\mathcal{B}_0$ is sufficient.

3 QUERY DESCRIPTION LANGUAGE Q_0

The query language Q_0 over an action signature Σ_0 consists of two types of expressions: axioms and queries. Axioms of Q_0 have a form

$$\text{initially}(l) \quad (5)$$

where l is a fluent literal. A collection of axioms describes the set of fluents known to be true in the initial situation. A set Γ of axioms is called *consistent* with respect to an action description $\mathcal{A}$ if the transition system defined by $\mathcal{A}$ has a state containing all l 's such that $\text{initially}(l) \in \Gamma$. Γ is called *complete* if for any fluent literal $\text{initially}(l)$ or $\text{initially}(\bar{l})$ is in Γ .

A query of Q_0 is a statement of the form

$$\text{holds_after}(l, \alpha) \quad (6)$$

where l is a fluent literal and α is a sequence of actions. The statement says that α can be executed in the initial situation and, if it were, then fluent literal l would necessarily be true afterwards. To give the semantics of Q_0 we need the following definitions:

Let T be a transition system over signature Σ_0 . We say that

- (i) a path $\sigma_0, a_1, \sigma_1, \dots, a_n, \sigma_n$ satisfies an axiom $\text{initially}(l)$ if $l \in \sigma_0$,
- (ii) a query $\text{holds_after}(l, [a_n, \dots, a_1])$ is a *consequence* of a set Γ of axioms in T if, for every path of T of the form $\sigma_0, a_1, \sigma_1, \dots, a_n, \sigma_n$ that satisfies all axioms in Γ , $l \in \sigma_n$.

Now we are ready for the main definition. Let $\mathcal{A}$ be an action description in some action description language over signature Σ_0 and T be the transition system described by $\mathcal{A}$. We say that a query Q is a *consequence* of a set Γ of axioms in $\mathcal{A}$ (symbolically, $\Gamma \models_{\mathcal{A}} Q$) if Q is a consequence of Γ in T .

In the next section we illustrate the use of $\mathcal{L}_0$ for the design of a plan checking system for the RCS.

4 THE RCS DOMAIN

The job of the RCS is primarily to provide maneuvering capabilities for the Space Shuttle while it is in orbit. When the RCS is functioning properly, or in cases of single failures, there are pre-scripted plans to accomplish any desired maneuver. Due to the huge number of combinations of failures that may occur, it is impossible to pre-script a plan for each multiple failure situation. If multiple failures occur during a mission, it is left up to mission controllers on the ground to develop the necessary plans. Time constraints and the serious repercussions of erroneous plans make a tool to help create and verify these plans extremely desirable. Such a tool could also be used by astronauts in case communication to their ground controllers were lost. Since astronauts have a wider, but much less in depth, knowledge of the shuttle's systems, the availability of a tool to help plan during a communications failure would greatly increase the chance of success. In this paper we describe a system used to verify plans. (Work on the planning part of the system is currently underway.)

The design of the system started with developing the action description for the RCS, $\mathcal{A}_{RCS}$, which contains information about the interconnection and function of its valves, jets, fuel tanks, electrical circuits, and switches. For illustrative purposes, we will focus on propositions from $\mathcal{A}_{RCS}$ which concern the switches. A more detailed description of the RCS domain model can be found in [19] and [5]. There are two types of switches in the RCS, each of which can be in several different positions. Each switch of the first type controls a pair of valves. Each switch of the second type controls an electrical circuit. In order for the shuttle to be able to perform a maneuver, one or more jets must be fired. The ability to fire these jets depends on the states of the valves and circuits, and therefore on the position of the switches. In the RCS domain there is only one type of action an agent can perform, changing the position of a switch. Performing an action of flipping a switch to a position causes the switch to be in the new position. For each switch, S , and each position, P , that the switch may be in, we will have the appropriate version of the following dynamic causal law.

$$\text{causes}(\text{flip}(S, P), \text{position}(S, P), []).$$

Flipping a switch to a given position also ensures that the switch is in no other position after performing the action. If $P1$ and $P2$ are two different positions then the dynamic causal law below describes this effect.

$causes(\text{flip}(S, P1), \neg \text{position}(S, P2), [])$.

Next we have a rule stating that it is impossible to flip a switch to a position it is already in.

$impossible(\text{flip}(S, P), [\text{position}(S, P)])$.

Note that, since there are 50 switches in the RCS subsystem, this rule cuts the number of executable actions in a situation from 100 down to 50. For any switch S and valve V controlled by S we have the following static causal law *OPEN VALVE*:

$causes([\text{position}(S, \text{open}),$
 $\neg \text{non_functional}(\text{open}, S),$
 $\neg \text{stuck}(\text{closed}, V)],$
 $\text{open}(V))$.

The law states that if the switch S is set to the open position and both S and V are functioning properly then V is open. *ARCS* also contains a similar causal law which states when the valve will be closed.

One may wonder why this was not represented by a dynamic causal law which stated that flipping the switch causes the valve to be open if the proper conditions were met. This can be explained by the following example.

Imagine we wish to model the operation of an ordinary lamp. One is tempted to have a dynamic causal law stating that if the switch is turned on, then the light comes on. But what if the bulb is burned out? We could add a precondition to the law stating that it only applies when the bulb is good. This, however, is only half the battle if we have an action to change the bulb. We would then need a dynamic causal law stating that changing the bulb causes the light to be on if the switch is on. Suppose we then update the domain by saying that the lamp can be unplugged and we add a new action to plug in the lamp. Both of the previous dynamic causal laws need to be updated and a new law needs to be added. Now consider a different approach. The dynamic causal laws simply state that turning the switch on causes the switch to be on and changing the bulb causes the bulb to be good. We then add a static causal law stating that if the switch is on and the bulb is good then the light is on. Now, in order to add the information about plugging in the lamp, we simply add a new dynamic causal law stating that plugging in the lamp causes it to be plugged in. We also must modify the one existing static causal law to reflect that the lamp must be plugged in for the light to be on. This approach is preferable for two primary reasons. First, as was shown by the example, it is more elaboration tolerant. The second reason deals with the initial situation. Using the first approach, we could have a consistent set of axioms which stated that the light was initially on and the bulb was initially burned out. Using the second approach, this initial situation is not consistent since it is not closed with respect to the static causal laws of the domain. Notice that the above argument suggests the second dynamic causal law above can be better written as a static causal law

$causes([P1 \neq P2, \text{position}(S, P1)], \neg \text{position}(S, P2))$.

This is indeed the case but we stay with the original representation since it substantially simplifies some of the proofs.

In our language, we also allow definition propositions. Certain circuits within the RCS must be switched on in order to operate. If X is such a circuit and S is the switch that controls it, then if the switch is on and functioning, then the circuit will be properly powered. This is captured by the following definition proposition *POWERED CIRCUIT*.

$definition(\text{powered}(X),$
 $[\text{position}(S, \text{on}), \neg \text{non_functional}(\text{on}, S)])$.

Note that this proposition is similar to the static causal law *OPEN VALVE*. A definition proposition is used since, unlike the law for *OPEN VALVE*, the head of *POWER CIRCUIT* holds *if and only if* the preconditions are met. This subtle difference can be illustrated by looking at the precondition that the switch be functional. In the case of the circuit, if the switch becomes non-functional while the circuit is powered, the circuit will no longer be powered. With the valve the situation is different. If the valve is already open and the switch fails, the valve does not close, it stays open.

After completion of the action description of the RCS we addressed the problem of computing the corresponding consequence relation which required knowledge of logic programming but no additional knowledge of the shuttle.

To make the paper self contained we include a brief introduction to the answer set semantics of logic programs [10] necessary for understanding of this step. The logic programming language used in the paper is substantially more powerful than the original Pure Prolog. In particular it allows both classical and nonmonotonic negations. For more detailed discussion of this language and its applications to knowledge representation see [3] and [12].

5 LOGIC PROGRAMS

The language of a logic program, like a first-order language, is determined by its signature, consisting of object constants, function constants and predicate constants. Terms are built as in the corresponding first-order language; positive literals (or atoms) have the form $p(t_1, \dots, t_n)$, where the t 's are terms and p is a predicate symbol of arity n ; negative literals are of the form $\neg p(t_1, \dots, t_n)$. The set of all ground literals over signature σ is denoted by $lit(\sigma)$. Literals of the form $p(t_1, \dots, t_n)$ and $\neg p(t_1, \dots, t_n)$ are called contrary. By $\bar{l}$ we denote a literal contrary to l . Literals and terms not containing variables are called ground. A rule is an expression of the form

$$l_0 \leftarrow l_1, \dots, l_m, \text{not } l_{m+1}, \dots, \text{not } l_n \quad (7)$$

where l_i 's are literals and *not* is a logical connective called *negation as failure* [7, 17]. The left-hand side of the rule is called the rule's head or conclusion; the right-hand side is called the rule's body (or premise). A pair $\{\sigma, \pi\}$ where σ is a signature and π is a collection of rules over σ is called a *logic program*.

Notice, that a statement *not l* is not a literal. A literal $\neg l$ stands for " l is false" while *not l* is informally read as "there is no reason to believe that l is true". In this section we assume that l 's in rule (7) are ground. Rules with variables (usually denoted by capital letters) will be used as shorthand for the sets of their ground instantiations.

The semantics of a logic program π assigns to π a collection of *answer sets* - sets of ground literals over signature $\sigma(\pi)$ of π corresponding to beliefs which can be held by a rational reasoner on the basis of rules of π . Under this semantics the rule (7) can be viewed as a constraint on such beliefs and is read as "if literals $l_1, \dots, l_m$ are believed to be true and there is no reason to believe that literals $l_{m+1}, \dots, l_n$ are true then the reasoner must believe l_0 ." We say that literal l is *true* in an answer set S of π if $l \in S$; l is *false* in S if $\bar{l} \in S$. l is *true* in π ($\pi \models l$) if l is true in all answer sets of π . We say that π 's answer to a query l is *yes* if $\pi \models l$, *no* if $\pi \models \bar{l}$, and *unknown* otherwise.

To give a definition of answer sets of logic programs, let us first consider programs without negation as failure.

- The *answer set* of program π not containing negation as failure *not* is the smallest (in the sense of set-theoretic inclusion) subset S of $lit(\sigma(\pi))$ such that

- (i) for any rule $l_0 \leftarrow l_1, \dots, l_m$ from π , if $l_1, \dots, l_m \in S$, then $l_0 \in S$;
- (ii) if S contains a pair of contrary literals, then $S = lit(\sigma(\pi))$.

It can be easily shown that every program π that does not contain negation as failure has a unique answer set which will be denoted by $ans(\pi)$.

- Now let π be an arbitrary logic program without variables. For any set S of literals, let π^S be the logic program obtained from π by deleting

- (i) each rule that has an occurrence of *not l* in its body with $l \in S$, and
- (ii) all occurrences of *not l* in the bodies of the remaining rules.

Clearly, π^S does not contain *not*, and hence its answer set is already defined. If this answer set coincides with S , then we say that S is an *answer set* of π . In other words, the answer sets of π are characterized by the equation

$$S = ans(\pi^S). \quad (8)$$

(For programs without classical negation the answer set semantics coincides with the stable model semantics of [9].) Notice, that the above definition of entailment in logic programs is purely declarative. There are different ways to compute this entailment. In particular, under certain conditions, a simple Prolog metainterpreter can be shown to be sound with respect to this entailment. The version of this metainterpreter based on a more modern logic programming language XSB [6] is also sound and provides an even better approximation.

6 TRANSLATION TO LOGIC PROGRAMMING

Our methodology for computing the consequence relation of $\mathcal{L}_0$ is a slight modification of a general approach suggested in [11], [4]. It is based on translating a domain description $\mathcal{D}$ (consisting of action description and axioms) into a logic program $\Pi(\mathcal{D})$ and reducing the computation of the consequence relation in $\mathcal{D}$ to answering queries in $\Pi(\mathcal{D})$. At the core of the translation is a collection of domain independent axioms formalizing reasoning about effects of actions. The development of these axioms was substantially influenced by two decades of research in nonmonotonic logics and semantics of logic programming. This research led to the methodology of representing and reasoning with defaults, i.e. statements of the form "normally, (typically, as a rule) elements of a class a have property p ". There are several defaults which are frequently used in reasoning about effects of actions. The most important one, known as the commonsense law of inertia [16], says that normally, things remain as they are. Any axiom describing the effect of an action on a state of the world represents an exception to this default. An agent reasoning about possible effects of his actions on the current state of the world uses these axioms to derive the changes that would occur in the current state after the execution of a particular action. The law of inertia is used to derive what does not change. The problem of constructing a formal framework which would allow us to express and reason with the law of inertia is called the frame problem. The use of negation as failure leads to a simple solution of the frame problem for a broad class of dynamic domains.

6.1 Domain independent axioms

In this section we outline the set of domain independent axioms, Π . We will assume that the program contains rules defining the following relations:

contrary(F, G) is true iff F and G are contrary fluent literals;

defined_literal(L) is true iff L occurs in the head of a definition from the corresponding action description;

frame_literal(F) iff F is a fluent literal which is neither a defined literal nor the negation of defined literal.

The next three rules define executable sequences of actions:

$$\begin{aligned} \text{impossible}([A \mid R]) \leftarrow \text{impossible}(A, P), \\ \text{hold_after}(P, R). \end{aligned}$$
$$\begin{aligned} \text{impossible}([A \mid R]) \leftarrow \text{impossible}(R). \\ \text{executable}(R) \leftarrow \text{not impossible}(R). \end{aligned}$$

Here $[A \mid R]$ is standard Prolog notation for the list with head A and tail R . (Recall that, since we execute actions in the list from right to left, A is the last action to be executed.) The first two rules state that a sequence of actions is impossible if either the last action in the sequence is impossible or if the rest of the sequence is impossible. The definition of executability relies on the completeness of our domain description. It says that if a sequence R of action is not known to be impossible then it is possible.

The next axiom determines what holds in the initial state of the domain.

$$\text{holds_after}(L, []) \leftarrow \text{initially}(L).$$

Here *initially*(L) is an axiom of $\mathcal{Q}_0$.

The next four rules determine the effects of causal laws of the corresponding action description.

$$\begin{aligned} \text{holds_after}(L, [A \mid R]) \leftarrow \text{causes}(A, L, P), \\ \text{hold_after}(P, R), \\ \text{executable}([A \mid R]). \end{aligned}$$

The rule says that a literal, L , holds as the result of performing an executable sequence of actions $[A \mid R]$, if the corresponding action description contains a dynamic causal law $\text{causes}(A, L, P)$ and all the preconditions from P hold after the execution of R .

$$\begin{aligned} \text{holds_after}(L, S) \leftarrow \text{causes}(G, L), \\ \text{hold_after}(G, S), \\ \text{executable}(S). \end{aligned}$$

The rule describes the effects of static causal laws.

The next two rules are concerned with definition propositions.

$$\text{holds_after}(L, S) \leftarrow \text{definition}(L, P), \\ \text{hold_after}(P, S), \\ \text{executable}(S).$$

$$\text{holds_after}(F, S) \leftarrow \text{contrary}(F, G), \\ \text{defined_literal}(G), \\ \text{not holds_after}(G, S), \\ \text{executable}(S).$$

The rules state that if there is a definition proposition with head L and body P and all the preconditions from P hold after the execution of S then L also holds. Otherwise, $\bar{L}$ holds.

The next pair of rules state when a set of literals hold.

$$\text{hold_after}([], -).$$

$$\text{hold_after}([H \mid T], A) \leftarrow \text{holds_after}(H, A), \\ \text{hold_after}(T, A).$$

The first says that the empty set of literals hold in any situation. The second states that a set of literals hold after a sequence of actions if each fluent in the set holds after that sequence of actions.

The commonsense law of inertia is captured by the following rule which states that fluents are normally not changed by performing actions. According to general methodology for representing defaults we use an "abnormality predicate" ab [15] to block the rule when an action does cause a change in the value of the fluent.

$$\text{holds_after}(L, [A \mid R]) \leftarrow \text{frame_literal}(L), \\ \text{holds_after}(L, R), \\ \text{not } ab(L, A, R), \\ \text{executable}([A \mid R]).$$

Note that the inertia rule applies only to frame fluents. The values of other fluents are fully determined by the rules for definition propositions.

Finally, the last two rules state that "a literal, L , is abnormal with respect to the inertia axiom if $\bar{L}$ was caused by either a dynamic or static causal law as a result of performing action, A , in the state that resulted from performing action sequence, R ."

$$ab(F, A, R) \leftarrow \text{contrary}(F, G), \\ \text{causes}(A, G, P), \\ \text{hold_after}(P, R).$$

$$ab(F, A, R) \leftarrow \text{contrary}(F, G), \\ \text{causes}(P, G), \\ \text{hold_after}(P, [A \mid R]).$$

6.2 Correctness and Usage

The correctness of the program Π with respect to the action description $\mathcal{A}_{RCS}$ is based on the general theorem about domain descriptions in $\mathcal{L}_0$ and on some properties of the action description $\mathcal{A}_{RCS}$ of the RCS domain.

Theorem 1 Let $\mathcal{D}$ be a domain description consisting of a deterministic and consistent action description $\mathcal{A}$ and a complete and consistent set of axioms Γ . Then for any query, $\text{holds_after}(l, \alpha)$, $\Gamma \models_{\mathcal{A}} \text{holds_after}(l, \alpha)$ iff $\mathcal{A} \cup \Pi \models \text{holds_after}(l, \alpha)$.

Proposition 1 The action description $\mathcal{A}_{RCS}$ is consistent and deterministic.

To actually execute the logic program $\pi = A \cup \Gamma \cup \Pi$ we need to have an interpreter capable of answering queries in logic programs with two negations. Such an interpreter, I , can be easily constructed on top of Prolog or XSB. To insure its correctness we need to show that, given a program π of the above form, the interpreter always terminates, does not require a so called occur check, does not flounder (i.e. does not attempt to prove a goal of the form *not* q where q contains uninstantiated variables), and satisfies several other simple properties. Fortunately, the theory of logic programming provides us with a comparatively simple way to check all these properties and to prove the following proposition:

Proposition 2 Let q be a ground query and $\pi = A_{RCS} \cup \Gamma \cup \Pi$ where Γ is a complete, consistent set of axioms. Then given π and q , the interpreter I answers yes iff $\pi \models q$.

These results establish correctness of our program with respect to A_{RCS} . In order to use the program, the flight controllers need to specify the current positions of the switches and valves, state the malfunctioning components, and provide other similar information which constitutes Γ . This requires knowledge of neither action description languages nor logic programming. If needed, consistency of the input can be checked automatically.

7 CONCLUSION AND FUTURE WORK

We believe that our experiment in the use of action languages was successful. The action language $\mathcal{L}_0$ has proven to be simple to use and understand. This was primarily seen in our communications with people from USA. We sent an early version of the RCS action description to our contact there (a former flight controller for the RCS with some knowledge of logic programming but no prior experience with actions languages). He was able to spot several errors simply by reading over the description. He also found the language intuitive enough that he has since written a preliminary, more technically detailed, domain description for another of the shuttle's subsystems, using the RCS action description as his only guide.

The mathematical theory of action languages and logic programming proved to be sufficiently developed to allow us to prove the properties of our system. As was intended, elements of logic programming were, for the most part, hidden from the end users. Logic programming, however, played a bigger role than expected during the formalization of the RCS domain. We also found that, in this domain, in order to properly specify some of our propositions, we needed to use recursive rules similar to that used in the definition of transitive closure. It remains to be seen if this can be avoided without a substantial complication of representation. So far we were not able to do that, which may point to the usefulness of logic programming languages even in the specification phase of the project.

Another interesting direction of research suggested by this experiment is the discovery of more powerful and preferably syntactic conditions which would guarantee consistency and determinism of an action specification. Ultimately, we wish to create an interface which would allow an expert in a given domain to create an action description with only minor knowledge of logic programming and permit end users to use the system with only knowledge of the interface.

In order to provide an even more useful tool, we are currently working to expand this system in several directions. Our first goal is to expand the system by adding a diagnostic component. This would be used when a sequence of actions was actually performed but unexpected results were observed. Using the action description, we would like the system to be able to determine what failure or failures may have occurred which would explain the observed behavior.

The second direction for further work concerns planning. While the current system can verify plans, it would be beneficial to be able to generate plans as well. We would like to be able to provide the system with the current situation and a goal (a set of axioms and a query from the language of $\mathcal{Q}_0$) and have it generate a plan to achieve the goal from that situation.

REFERENCES

- [1] Alferes, J.J. and Pereira, L.M. *Reasoning with Logic Programming*. Lecture Notes in Artificial Intelligence, Springer, 1996.
- [2] Apt, K. and Bol, R. Logic programming and negation: a survey. In *Journal of Logic Programming*, Vol. 19/20, pp. 9-71, 1994.

- [3] Baral, C. and Gelfond, M. Logic programming and knowledge representation. *Journal of Logic Programming*, Vol. 12, pp. 1-80, 1994.
- [4] Baral C., Gelfond, M., and Proveti, A. Representing Actions: Laws, Observations, and Hypothesis. In *Journal of Logic Programming*, Vol. 31, No. 1-3, pp. 245-298, 1997.
- [5] Barry, M. and Watson, R. Reasoning about actions for spacecraft redundancy management. In *Proc. of the 1999 IEEE Aerospace Conference*, to appear.
- [6] Chen, W., Swift, T., and Warren, D. Efficient top-down computation of queries under the well-founded semantics. In *Journal of Logic Programming*, Vol. 24, No. 3, pp. 161-201, 1995.
- [7] Clark, K. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293-322. Plenum Press, New York, 1978.
- [8] Electronic Transaction on Artificial Intelligence. Web Site, On-line at <<http://www.ida.liu.se/ext/etai/index>>.
- [9] Gelfond, M. and Lifschitz, V. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1070-1080, 1988.
- [10] Gelfond, M. and Lifschitz, V. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365-387, 1991.
- [11] Gelfond, M. and Lifschitz, V. Representing Actions in Extended Logic Programs. In *Proc. of Joint International Conference and Symposium on Logic Programming*, pp. 559-573, 1992.
- [12] Lifschitz, V. Foundations of logic programming. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 69-128. CSLI Publications, 1996.
- [13] Lifschitz, V., editor. *Journal of Logic Programming - Special Issue: Reasoning about Action and Change*, Elsevier, 1997.
- [14] McCain, N. and Turner, H. A causal theory of ramifications and qualifications. In *Proc. of IJCAI 95*, pp. 1978-1984, 1995.
- [15] McCarthy, J. Applications of circumscription to formalizing common sense knowledge. In *Artificial Intelligence*, 26(3), pp. 89-116, 1986.
- [16] McCarthy, J. and Hayes, P. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pp. 463-502. Edinburgh University Press, Edinburgh, 1969.
- [17] Reiter, R. On closed world data bases. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 119-140. Plenum Press, New York, 1978.
- [18] Turner, H. Representing actions in logic programs and default theories: A situation calculus approach. In *Journal of Logic Programming*, Vol. 31, No. 1-3, pp. 245-298, 1997.
- [19] Watson, R. An application of action theory to the space shuttle. In *Proc. of Practical Applications of Declarative Languages '99*, to appear.

The role of observations in probabilistic open systems

Murali Narasimha*

Rance Cleaveland†

Purushothaman Iyer*

Abstract

This paper considers a logic, based on the modal μ -calculus, for describing properties of probabilistic open distributed systems and develops a model-checking algorithm for determining whether or not states in finite-state probabilistic systems satisfy formulas in the logic. The central contribution of the paper is a semantics that distinguishes between observations, the meaning of a temporal formula, and its measure. The ensuing model-checking problem reduces to the calculation of a (particular) solution to a system of non-linear equations.

1 Introduction

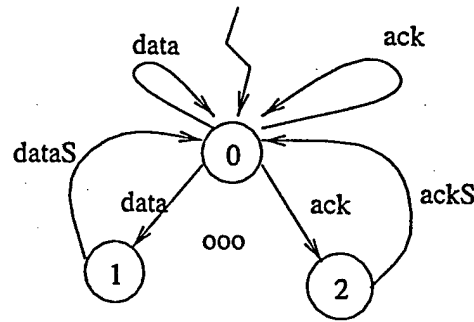
The era of net-centric computing is here, fueled by easy to use applications. In the near future the number of network-based applications is expected to grow exponentially. These applications mix audio, video and text and consequently make great demands on the network traffic. Consequently, the eventual success of these applications will depend upon quality of service (QoS) guarantees that can be provided to the end-users. Not coincidentally, military applications (such as command and control) have similar, though even more stringent, service requirements. Use of formal methods for developing and checking design specifications of concurrent systems and for conformance testing of the implementations has gained currency over the past decade. There have been several success stories reported in the literature [5]. However, these mathematics based techniques have been restricted to reasoning about qualitative, i.e., functional aspects of distributed system. In this note we will consider how specifications of open distributed systems may be structured and how QoS requirements of such systems may be stated. The main contribution of this paper is a novel technique for describing the semantics of open distributed system specifications containing probabilistic information. Our semantic technique allows a precise calculation of the probability with which a temporal property of an open distributed system holds. The paper is organized as follows: in Section 2 we briefly describe the specification and requirements language for non-probabilistic open distributed systems. In Sections 3 and 4 we show how the specification and the requirements language from Section 2 can be extended to probabilistic open distributed systems. We follow that with a comparison of our semantics with extant work. In Sections 6 and 7 we discuss probabilistic model-checking and our goals for future work.

2 Specifying open distributed systems and their requirements

In general, the literature on concurrent systems distinguishes between *open systems* and *closed systems*. The former may require interaction with their environments in order to make progress; the latter are self-contained. Semantically, the difference between these kinds of systems is reflected in the mathematical models developed for them. Open systems are often represented using *labeled transition systems* which may be thought of as finite-state machines whose transition labels represent capabilities for interaction with the environment and which are used as mathematical entities to provide semantics to calculi based on process algebras such as CCS and CSP [6, 12, 1]. Closed systems, on the other hand, are usually modeled using *Kripke structures* which may be thought of as node-labeled directed graphs whose vertices correspond to system states and whose edges represent execution steps. The vertex labels contain information that is true of the state. Typical examples of open systems include communication protocols which require a user to invoke a service primitive before engaging in any activity. Closed systems include control systems in which a controller and the process being monitored interact only with each other.

*Dept of Computer Science, North Carolina State University, Raleigh, NC 27695.

†Dept of Computer Science, State University of New York, Stony Brook, NY 11794-4400.



dataS = Send data
 data = Receive data

 ack = Receive Ack
 ackS = Send Ack

Figure 1: A lossy half-duplex line

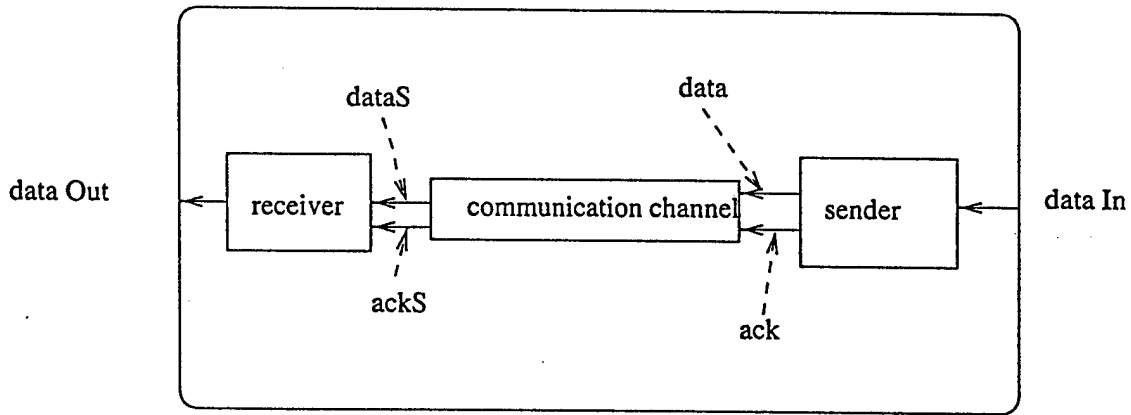


Figure 2: An architectural view of a communication system

Consider for example the communication medium of a network system. A half-duplex line which takes messages at one end and (sometimes) delivers it at the other end can be succinctly represented by the finite state machine in Figure 1. The self-loop transition from state 0 to state 0 labelled data models the act of the communication medium which receives a data message from the environment and drops it. The sequence of transitions from state 0 to 1 and back to 0 characterizes the behavior of the medium which accepts a message from its environment and faithfully delivers it. As can be observed the machine responds to input (data or ack) from the environment. It is this notion of external control or external non-determinism that characterizes the open system model.

Clearly the communication medium is merely a part of a whole system which also includes a sender and a receiver. One can obtain a system by composing the three processes together. The three sub-systems now act in concert with each other and present the view of a single system to an external observer (see Figure 2). Assuming that the sender and the receiver act as an intermediary between the user and the communication medium the behavior of the entire system is equivalent to the behavior of the communication medium depicted in Figure 1.

Formally Labelled transition systems are defined as follows:

Definition 2.1 A Labelled Transition System (LTS) $L = (S, Act, \rightarrow)$ where S is a set of states, Act is the set of actions that the system L may engage in and $\rightarrow \subseteq S \times Act \times S$ is the transition relation expressing permissible actions of the system L .

2.1 A logic for stating requirements

Temporal logics have traditionally been used to reason about *closed* systems; more precisely Kripke structures (see above) have been used as models for these logics. The μ -calculus [9] on the other hand permits properties of open systems (labeled transition systems) to be formulated. The logic is very sparse syntactically; in addition to the usual boolean operators it includes modalities indexed by sets of transition labels and schemes for defining formulas recursively. Thus for example a state in a labeled transition system satisfies $[S]\phi$ if every transition from the state labeled by an element of S satisfies ϕ and it satisfies $\langle S \rangle \phi$ if it has some transition labeled by an element of S that satisfies ϕ . Recursive formulas have the form $\mu X. \phi$ or $\nu X. \phi$. The former represents the “least” solution to the “equation” $X = \phi$ and is useful in expressing liveness properties; the latter corresponds to the “largest” solution to the same equation and is used in formulating safety properties. The appeal of the μ -calculus lies in its expressive power and its ability to encode many other temporal logics in a uniform fashion [2, 4, 3]; this power results from its support for defining recursive propositions. The main disadvantage of the μ -calculus is that formulas can be difficult to interpret owing to the complexity that can result from mutually recursive definitions. This needn’t be a problem in practice however as users can define the properties of interest in a higher-level notation with tools then handling the translation into the μ -calculus for the purpose of automated analysis.

Continuing with the example of the faulty half-duplex line one of the requirements that we may wish to impose is that there are no deadlocks in the system. This can be captured using the formula $\mu X. (_) tt \wedge [_] X$ (where $_$ represents the set of all labels) which is true of a state s provided there is at least one transition out of state s and the formula is true of every successor state of s . It is easy to show that this property is true of our example system.

Consider another property – that of eventual delivery of messages – which states that it is impossible to send an infinite sequence of data packets without any of it being delivered. This property can be captured by the formula $(\neg \nu X. (\text{data}) X)$. However this property does not hold true of our system as indeed there is an execution sequence where all data packets sent are lost. However truth or falsity of this property is of no practical value; it is generally expected that systems do fail (even communication systems) and it is relative rate of failure that is more important. In the rest of the paper we will show how to model and reason about such reliability issues.

3 Probabilistic Transition Systems

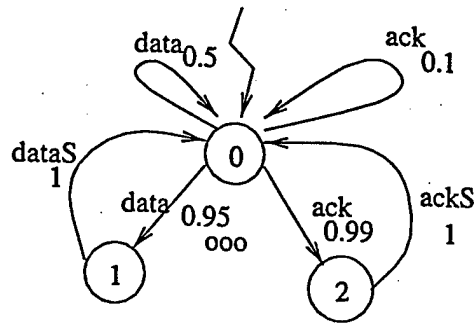
In this section we will introduce the probabilistic reactive system model which could be used for specifying and reasoning about faulty communication medium (as in our example). *Reactive probabilistic labeled transition systems* (PLTS for short) of [15, 10] are models of probabilistic computation. These are defined with respect to fixed sets *Act* and *Prop* of atomic actions and propositions respectively. The former set records the interactions the system may engage in with its environment while the latter provides information about the states the system may enter.

Definition 3.1 A PLTS L is a tuple (S, δ, P, I) where

- $(s, s', s_1 \in S)$ is a countable set of states;
- $\delta \subseteq S \times \text{Act} \times S$ is the transition relation;
- $P : \delta \rightarrow (0, 1]$ the transition probability distribution satisfies: $\sum_{(s, a, s') \in \delta} P(s, a, s') \in \{0, 1\}$ for all $s \in S$ and $a \in \text{Act}$; and
- $I : S \rightarrow 2^{\text{Prop}}$ is the interpretation function.

Intuitively a PLTS records the operational behavior of a system with S representing the possible system states and δ the execution steps enabled in different system states; each such step is labeled with an action and the intention is that when the environment of the system enables the action the system may engage in a transition labeled by the action. When this is the case $P(s, a, s')$ represents the probability with which the transition (s, a, s') is selected as opposed to other transitions labeled by a emanating from state s . Note that the conditions on P ensure that if $(s, a, s') \in \delta$ for some s' then $\sum_{(s, a, s') \in \delta} P(s, a, s') = 1$. In what follows we write $s \xrightarrow{a} s'$ if $(s, a, s') \in \delta$.

Considering our running example of a faulty medium we could specify that 5% of all data packets are lost by the communication medium while only 1% of the ack packets are lost. The difference in error rate could be due to



dataS = Send data
data = Receive data

ack = Receive Ack
ackS = Send Ack

Figure 3: A probabilistic characterization of faults in a lossy medium

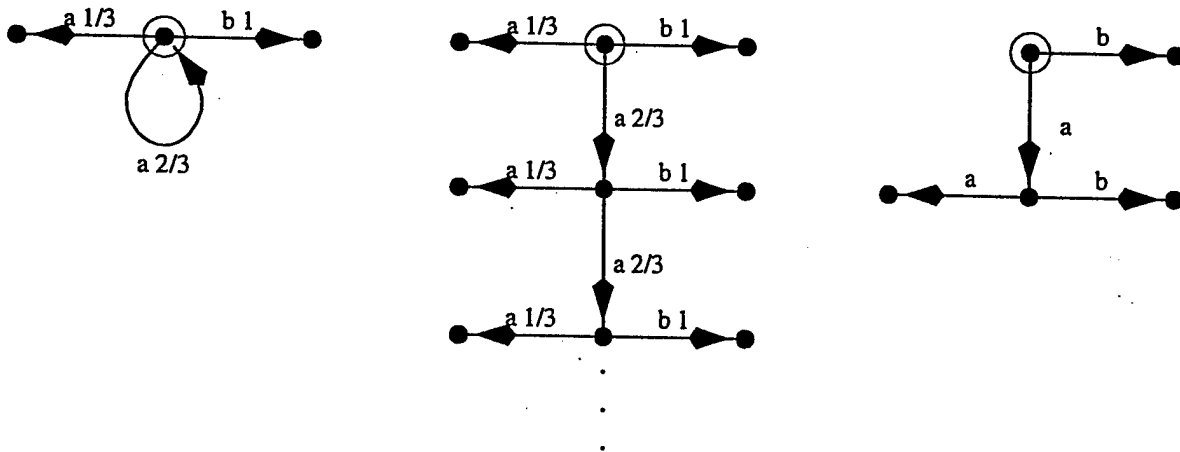


Figure 4: A PLTS's unrolling from a state and an observation.

the fact that data packets are traditionally longer and thus have a greater chance of being corrupted. The modified specification is given in Figure 3.

Given such a specification we might wish to check whether it satisfies the following requirement:

It is always true that the probability of successfully sending a data packet in three tries is greater than 99% and that the probability of successfully sending an ack packet in two tries is greater than 99.5%.

To answer this question we will have to describe a measure space over which our logical specifications are interpreted. To that end we wish to view a (state in a) PTL as an "experiment" in the probabilistic sense with an "outcome" or "observation" representing a resolution of all the possible probabilistic choices of transitions the system might experience as it executes. More specifically given a state in the PLTS we can unroll the PLTS into an infinite tree rooted at this state. An observation would then be obtained from this tree by resolving all probabilistic choices i.e. by removing all but one edge for any given action from each node in the tree. Figure 4 presents a sample PLTS's unrolling from a given state and an associated observation.

3.1 PLTSs and Measure Spaces of Observations

To define the observation trees of a PLTS we introduce *partial computations* which will form the nodes of the trees.

Definition 3.2 Let $L = (S, \delta, P, I)$ be a PLTS. Then a sequence of the form $s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_n} s_n$ is a *partial computation* of L if $n \geq 0$ and for all $0 \leq i < n$ $s_i \xrightarrow{a_{i+1}} s_{i+1}$.

Note that any $s \in S$ is a partial computation. If $\sigma = s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_n} s_n$ is a partial computation then we define $\text{fst}(\sigma)$ to be s_0 and $\text{lst}(\sigma)$ to be s_n . We also use $(\sigma, \sigma') \in C_L$ to refer to the set of all partial computations of L and take $C_L(s) = \{\sigma \in C_L \mid \text{fst}(\sigma) = s\}$ for $s \in S$. We define the following notations for partial computations.

Definition 3.3 Let $\sigma = s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_n} s_n$ and $\sigma' = s'_0 \xrightarrow{a'_1} s'_1 \dots \xrightarrow{a'_n} s'_n$ be partial computations of PLTS $L = (S, \delta, P, I)$ and let $a \in \text{Act}$.

1. If $s_n \xrightarrow{a} s'_0$ then $\sigma \xrightarrow{a} \sigma'$ is the partial computation $s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_n} s_n \xrightarrow{a} s'_0 \xrightarrow{a'_1} s'_1 \dots \xrightarrow{a'_n} s'_n$.
2. σ' is a *prefix* of σ if $\sigma' = s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_i} s_i$ for some $i \leq n$.

We also introduce the following terminology for sets of partial computations.

Definition 3.4 Let $L = (S, \delta, P, I)$ be a PLTS and let $C \subseteq C_L$ be a set of computations.

1. C is *prefix-closed* if for every $\sigma \in C$ and σ' a prefix of σ $\sigma' \in C$.
2. C is *deterministic* if for every $\sigma, \sigma' \in C$ with $\sigma = s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_n} s_n \xrightarrow{a} s \dots$ and $\sigma' = s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_n} s_n \xrightarrow{a'} s' \dots$ either $a \neq a'$ or $s = s'$.

The term prefix-closed is standard but the notion of determinacy of sets of partial computations deserves some comment. Intuitively if two computations in a deterministic set of partial computations share a common prefix then the first difference they can exhibit must involve transitions labeled by different actions; they cannot involve different transitions with the same action label.

We can now define the deterministic trees or *d-trees* of a PLTS L as follows.

Definition 3.5 Let $L = (S, \delta, P, I)$ be a PLTS. Then $\emptyset \neq T \subseteq C_L$ is a *d-tree* if the following hold.

1. There exists an $s \in S$ such that $T \subseteq C_L(s)$.
2. T is prefix-closed.
3. T is deterministic.

If C is a d-tree then we use $\text{root}(C)$ to refer to the s such that $C \subseteq C_L(s)$ and $\text{edges}(C)$ to refer to the relation $\{(\sigma, a, \sigma') \mid \sigma, \sigma' \in C \wedge \exists s' \in S. \sigma' = \sigma \xrightarrow{a} s'\}$.

We use $\mathcal{T}_L$ to refer to all the d-trees of L and set $\mathcal{T}_L(s) = \{T \in \mathcal{T}_L \mid \text{root}(T) = s\}$. We call T' a *prefix* of T if $T' \subseteq T$. We write $T \xrightarrow{a} T'$ if $\{\text{root}(T) \xrightarrow{a} \sigma' \mid \sigma' \in T'\} \subseteq T$; intuitively T' is then the subtree of T pointed to by an a -labeled edge. A d-tree T is *finite* if $|T| < \infty$. Finally we say that a d-tree is *maximal* if there exists no d-tree T' with $T \subset T'$ and use $\mathcal{M}_L$ and $\mathcal{M}_L(s)$ to refer to the set of all maximal d-trees of L and all maximal d-trees of L rooted at s respectively.

We wish to view the maximal deterministic d-trees of a PLTS as the “outcomes” of the PLTS and to talk about the likelihoods of different sets of outcomes. In order to do this we define a probability space over maximal d-trees rooted at a given state of L . The construction of this space is very similar in spirit to the standard sequence space construction for Markov chains [8]: we define a collection of “basic cylindrical sets” of maximal trees and use them to build a probability space over sets of maximal trees. The technical details appear below; in what follows fix $L = (S, \delta, P, I)$.

A *basic cylindrical subset* of $\mathcal{M}_L(s)$ contains all trees sharing a given finite prefix.

Definition 3.6 Let $s \in S$ and let $T \in \mathcal{T}_L(s)$ be finite. Then $B_T \subseteq \mathcal{M}_L(s)$ is defined as: $B_T = \{T' \in \mathcal{M}_L \mid T \subseteq T'\}$.

We can also define the *measure* of a basic cylindrical set as follows.

Definition 3.7 Let $T \in \mathcal{T}_L(s)$ be finite and let B_T be the associated basic cylindrical set. Then the *measure* $m(B_T)$ of B_T is given by: $m(B_T) = \prod_{(\sigma, a, \sigma') \in \text{edges}(T)} P(\text{lst}(\sigma), a, \text{lst}(\sigma'))$.

Intuitively $m(B_T)$ represents the proportion of all maximal d-trees emanating from the root of B_T that have B_T as a prefix.

For any given state s in L we can form the associated collection of basic cylindrical sets B_s^- consisting of sets of the form B_T for finite T with $\text{root}(T) = s$. We can then define a probability space $(\mathcal{M}_L(s), B_s, m_s)$ as follows.

Definition 3.8 Let $s \in S$. Then B_s is the smallest field of sets containing B_s^- and closed with respect to denumerable unions and complementation. $m_s : B_s \rightarrow [0, 1]$ is then defined inductively as follows.

$$\begin{aligned} m_s(B_T) &= m(B_T) \\ m_s\left(\bigcup_{i \in I} B_i\right) &= \sum_{i \in I} m_s(B_i) \text{ for pairwise disjoint } B_i \\ m_s(B^c) &= 1 - m_s(B) \end{aligned}$$

It is easy to show that for any s m_s is a probability measure over B_s . Consequently $(\mathcal{M}_L(s), B_s, m_s)$ is indeed a probability space. We refer to a set $M \subseteq \mathcal{M}_L(s)$ as *measurable* if $M \in B_s$.

4 Syntax of GPL

Generalized Probabilistic Logic (GPL) is parameterized with respect to a set $(X, Y \in) \text{Var}$ of propositional variables¹ a set $(a, b \in) \text{Act}$ of actions and a set $(A \in) \text{Prop}$ be a set of atomic propositions. The syntax of GPL may then be given using the following BNF-like grammar where $0 \leq p \leq 1$.

$$\begin{aligned} \phi &::= A \mid \neg A \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \mathbb{E}_{>p} \psi \mid \mathbb{E}_{\geq p} \psi \\ \psi &::= \phi \mid X \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \langle a \rangle \psi \mid [a] \psi \mid \mu X. \psi \mid \nu X. \psi \end{aligned}$$

The operators μ and ν bind variables in the usual sense and one may define the standard notions of free and bound variables. Also we refer to an occurrence of a bound variable X in a formula as a μ -occurrence if the closest enclosing binding operator for X is μ and as a ν -occurrence otherwise. GPL formulas are required to satisfy the following additional restrictions: they must contain no free variables and no sub-formula of the form $\mu X. \psi$ ($\nu X. \psi$) may contain a free ν -occurrence (μ -occurrence) of a variable.¹ In what follows we refer to formulas generated from nonterminal ϕ etc. as *state formulas* and those generated from ψ as *fuzzy formulas*; the formulas of GPL are the state formulas. We use $(\phi, \phi' \in) \Phi$ to represent the set of all state formulas and $(\psi, \psi' \in) \Psi$ for the set of all fuzzy formulas. In the remainder of the paper we write $\gamma[\Gamma'/X]$ to denote the simultaneous substitution of γ' for all free occurrences of X in γ . We also note that although the logic limits the application of $\neg$ to atomic propositions this does not restrict the expressiveness of the logic as we indicate later.

The next subsection defines the formal semantics of GPL but the intuitive meanings of the operators may be understood as follows. Fuzzy formulas are to be interpreted as specifying sets of *observations* of PLTSs which are themselves non-probabilistic trees as discussed above. An observation is in the set corresponding to the fuzzy formula if the root node of the observation satisfies the formula interpreted as a traditional mu-calculus formula: so $\langle a \rangle \psi$ holds of an observation if the root has an a -transition leading to the root of an observation satisfying ψ while it satisfies $[a] \psi$ if every a -transition leads to such an observation. Conjunction and disjunction have their usual interpretation. $\mu X. \psi$ and $\nu X. \psi$ are fixpoint operators describing the “least” and “greatest” solutions respectively to the “equation” $X = \psi$. It will turn out that any state in a given PLTS defines a probability space over observations and that our syntactic restrictions ensure that the sets of observations defined by any fuzzy formula are *measurable* in a precise sense. State formulas will then be interpreted with respect to states in PLTSs with a state satisfying a formula of the form $\mathbb{E}_{\geq p} \psi$ if the measure of observations corresponding to the state is at least p .

4.1 Semantics of Fuzzy Formulas

In the remainder of this section we define the semantics of GPL formulas with respect to a fixed PLTS $L = (S, \delta, P, I)$ by giving mutually recursive definitions of a relation $\models_L \subseteq S \times \Phi$ and a function $\Theta_L : \Psi \rightarrow 2^{\mathcal{M}_L}$. The former

¹In other words, formulas must be alternation-free in the sense of [3].

indicates when a state satisfies a state formula while the latter returns the set of maximal d-trees satisfying a given fuzzy formula. In this subsection we present Θ_L ; the next subsection then considers $\models_L$. In what follows we fix $L = (S, \delta, P, I)$.

Our intention in defining $\Theta_L(\psi)$ is that it return trees that interpreted as (non-probabilistic) labeled transition systems satisfy ψ interpreted as a mu-calculus formula. To this end we augment Θ_L with an extra environment parameter $e : Var \rightarrow 2^{\mathcal{M}_L}$ that is used to interpret free variables. The formal definition of Θ_L is the following.

Definition 4.1 The function Θ_L is defined inductively as follows.

- $\Theta_L(\phi)e = \bigcup_{s \models_L \phi} \mathcal{M}_L(s)$
- $\Theta_L(X)e = e(X)$
- $\Theta_L(\langle a \rangle \psi)e = \{T \mid \exists T' : T \xrightarrow{a} T' \wedge T' \in \Theta_L(\psi)e\}$
- $\Theta_L([a]\psi)e = \{T \mid (T \xrightarrow{a} T') \Rightarrow T' \in \Theta_L(\psi)e\}$
- $\Theta_L(\psi_1 \wedge \psi_2)e = \Theta_L(\psi_1)e \cap \Theta_L(\psi_2)e$
- $\Theta_L(\psi_1 \vee \psi_2)e = \Theta_L(\psi_1)e \cup \Theta_L(\psi_2)e$
- $\Theta_L(\mu X. \psi)e = \bigcup_{i=0}^{\infty} M_i$ where $M_0 = \emptyset$ and $M_{i+1} = \Theta_L(\psi)e[X \mapsto M_i]$.
- $\Theta_L(\nu X. \psi)e = \bigcap_{i=0}^{\infty} N_i$ where $N_0 = \mathcal{M}_L$ and $N_{i+1} = \Theta_L(\psi)e[X \mapsto N_i]$.

When ψ has no free variables $\Theta(\psi)e = \Theta(\psi)e'$ for any environments e, e' . In this case we drop the environment e and write $\Theta_L(\psi)$.

Some comments about this definition are in order. Firstly it is straightforward to show that the semantics of all the operators except μ and ν are those that would be obtained by interpreting maximal deterministic trees as labeled transition systems and fuzzy formulas as mu-calculus formulas in the usual style [9]. Secondly because d-trees are deterministic it follows that if $T \in \Theta_L(\langle a \rangle \psi)$ then $T \in \Theta_L([a]\psi)$. Finally the definitions we have given for μ and ν differ from the more general accounts that rely on the Tarski-Knaster fixpoint theorem. However because of the “alternation-free” restriction we impose on our logic and the fact that d-trees are deterministic the meanings of $\mu X. \psi$ and $\nu X. \psi$ are still least and greatest fixpoints in the usual sense.

We close this section by remarking on an important property of Θ_L . For a given $s \in S$ let $\Theta_{L,s}(\psi) = \Theta_L(\psi) \cap \mathcal{M}_L(s)$ be the maximal d-trees from s “satisfying” ψ . We have the following.

Theorem 4.2 For any $s \in S$ and $\psi \in \Psi$, $\Theta_{L,s}(\psi)$ is measurable.

4.2 Semantics of State Formulas

We now define the semantics of state formulas by defining the relation $\models_L$.

Definition 4.3 Let $L = (S, \delta, P, I)$ be a PLTS. Then $\models_L$ is defined inductively as follows.

- $s \models_L A$ iff $A \in I(s)$.
- $s \models_L \neg A$ iff $A \notin I(s)$.
- $s \models_L \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$.
- $s \models_L \phi_1 \vee \phi_2$ iff $s \models \phi_1$ or $s \models \phi_2$.
- $s \models_L \mathcal{A}_{>p} \psi$ iff $m_s(\Theta_{L,s}(\psi)e) > p$.
- $s \models_L \mathcal{A}_{\geq p} \psi$ iff $m_s(\Theta_{L,s}(\psi)e) \geq p$.

An atomic proposition is satisfied by a state if the proposition is a member of the propositional labeling of the state. Conjunction and disjunction are interpreted in the usual manner while a state satisfies a formula $\mathcal{A}_{>p} \psi$ iff the measure of the observations of ψ rooted at s exceeds p and similarly for $\mathcal{A}_{\geq p} \psi$.

4.2.1 Properties of the Semantics

We close this section by remarking on some of the properties of GPL. The first shows that the modal operators for fuzzy formulas enjoy certain distributivity laws with respect to the propositional operators.

Lemma 4.4 *For a PLTS L , fuzzy formulas ψ_1 and ψ_2 and $a \in \text{Act}$, we have:*

1. $\Theta_L(\langle a \rangle(\psi_1 \vee \psi_2)) = \Theta_L(\langle a \rangle\psi_1 \vee \langle a \rangle\psi_2)$
2. $\Theta_L([a](\psi_1 \vee \psi_2)) = \Theta_L([a]\psi_1 \vee [a]\psi_2)$
3. $\Theta_L(\langle a \rangle(\psi_1 \wedge \psi_2)) = \Theta_L(\langle a \rangle\psi_1 \wedge \langle a \rangle\psi_2)$
4. $\Theta_L([a](\psi_1 \wedge \psi_2)) = \Theta_L([a]\psi_1 \wedge [a]\psi_2)$
5. $\Theta_L([a]\psi_1 \wedge \langle a \rangle\psi_2) = \Theta_L(\langle a \rangle(\psi_1 \wedge \psi_2))$

That $[a]$ distributes over $\vee$ and $\langle a \rangle$ over $\wedge$ is due to the determinacy of d-trees.

Based on Theorem 4.2 and the definition of Θ_L the next lemma also holds.

Lemma 4.5 *Let $s \in S$, $a \in \text{Act}$ and $\psi, \psi_1, \psi_2 \in \Psi$. Then we have the following.*

$$m_s(\Theta_L(\psi_1 \vee \psi_2)) = m_s(\Theta_L(\psi_1)) + m_s(\Theta_L(\psi_2)) - m_s(\Theta_L(\psi_1 \wedge \psi_2)) \quad (1)$$

$$m_s(\Theta_L(\langle a \rangle\psi)) = \sum_{(s,a,s') \in \delta} P(s,a,s') * m_{s'}(\Theta_L(\psi)) \quad (2)$$

$$m_s(\Theta_L([a]\psi)) = \begin{cases} m_s(\Theta_L(\langle a \rangle\psi)) & \text{if } (s,a,s') \in \delta \text{ for some } s' \\ 1 & \text{otherwise} \end{cases} \quad (3)$$

Finally although our logic only allows a restricted form of negation we do have the following.

Lemma 4.6 *Let $L = (S, \delta, P, I)$ be a PLTS with $s \in S$, and let ψ and ϕ be fuzzy and state formulas, respectively. Then there exist formulas $\text{neg}(\psi)$ and $\text{neg}(\phi)$ such that:*

$$\Theta_{L,s}(\text{neg}(\psi)) = \mathcal{M}_L(s) - \Theta_{L,s}(\psi) \quad \text{and} \quad s \models_L \text{neg}(\phi) \iff s \not\models_L \phi.$$

Proof 1 Follows from the duality of $\wedge/\vee$, $\Gamma[a]/\langle a \rangle$, $\Gamma\nu/\mu$ and $\mathbb{E}_{>p}/\mathbb{E}_{\geq 1-p}$. ■

5 Expressiveness of GPL

In this section we will compare our interpretation of GPL with a similar effort by Huth and Kwiatkowska [7] who develop a notion of *quantitative model checking* [7] in which one calculates the likelihood with which a system state satisfies a formula. The basis for their approach lies in a semantics for the modal mu-calculus that assigns “probabilities” rather than truth values to assertions about states in a PLTS. In this section we briefly review their approach offer a criticism of it and show how GPL provides a principled means of remedying the criticism.

The syntax of their logic coincides with the semantics of our fuzzy formulas with the following exceptions: (1) they allow negation (although in such a way that negations can be eliminated in the usual manner); (2) the only atomic propositions are tt (“true”) and ff (“false”); (3) no use of the probabilistic quantifiers $\mathbb{E}_{\geq p}$ and $\mathbb{E}_{>p}$ is allowed. They then present three semantics for the logic that differ only in their interpretation of conjunction. Each interprets formulas as functions mapping states to numbers in $[0, 1]$; formally given PLTS L $\llbracket \psi \rrbracket_L : S \rightarrow [0, 1]$ represents the interpretation of formula ψ . What follows presents the relevant portions of these semantics.

$$\begin{aligned} \llbracket \text{tt} \rrbracket_L(s) &= 1 \\ \llbracket \langle a \rangle \psi \rrbracket_L(s) &= \sum_{s' \in \delta(s,a)} P(s,a,s') \cdot \llbracket \psi \rrbracket_L(s') \\ \llbracket \psi_1 \wedge \psi_2 \rrbracket_L(s) &= f(\llbracket \psi_1 \rrbracket_L(s), \llbracket \psi_2 \rrbracket_L(s)) \end{aligned}$$

The meanings of the other boolean and modal operators may be obtained using dualities (e.g. $\llbracket [a]\psi \rrbracket_L(s) = 1 - \llbracket \langle a \rangle \neg \psi \rrbracket_L(s)$) while the meanings of fixed points may be obtained using the usual Tarski-Knaster construction. The semantics of $\wedge$ contains a parameter f ; [7] provides three different instantiations of f .

1. $f(x, y) = \min(x, y)$
2. $f(x, y) = x \cdot y$
3. $f(x, y) = \max(x + y - 1, 0)$

Each unfortunately has its drawbacks. The first two fail to validate some expected logical equivalences; for example it is not the case that it is equivalent to $\psi \vee \neg \psi$. The authors refer to the third as a “fuzzy” interpretation and indicate that it is intended only to provide a “lower approximation” on probabilities; “real” probabilities are therefore not calculated.

GPL permits a similar interpretation to be attached to the mu-calculus but in such a way that exact probabilities may be assigned to formulas. Consider the function $\llbracket \psi \rrbracket_L^{GPL}$ given by:

$$\llbracket \psi \rrbracket_L^{GPL}(s) = m_s(\Theta_L(\psi)).$$

One can show that this interpretation preserves much of the semantics of Huth and Kwiatkowska; in particular Lemmas 4.5 and 4.6 show that this definition attaches the same interpretations to the modalities. It is also the case that expected logical equivalences hold and that this interpretation yields a probability with a precise measure-theoretic interpretation. Finally it should be easy to observe that our logic coincides with probabilistic bisimulation [10] – a property not true of Huth and Kwiatkowska’s interpretation.

6 Model Checking

We will now provide a brief description of the model-checking procedure; complete details can be found in [11]. Much like traditional model-checking procedures our model-checking procedure works bottom up by considering the smallest possible state formula and then working up. The only non-trivial state formula are those of the form $\mathbb{E}_{\geq p} \psi$. To check whether $\mathbb{E}_{\geq p} \psi$ holds of a state s_0 we need to compute the set of s_0 -rooted trees that satisfy $\mathbb{E}_{\geq p} \psi$ and then calculate its measure. However this two step approach is unworkable. What we do instead is build a dependency graph that allows us to state equations expressing $m_{s_0}(\Theta_L(\psi))$ in terms of $m_{s'}(\Theta_L(\wedge F))$ where s' are states reachable from s_0 and F contains subformulae of ψ . Consequently the nodes of the dependency graph are labelled by pairs of the form (s, F) where s is a state of the PLTS and F is a subset of the subformulae of ψ . The edges that connect these nodes capture the dependency. For instance if F contains a formula of the form $\phi = \phi_1 \wedge \phi_2$ then there is a unique successor node in the dependency graph labelled by $(s, F - \{\phi\} \cup \{\phi_1, \phi_2\})$. Since a node (s, F) is intended to characterize $m_s(\Theta_L(\wedge F))$ it should be clear that the measures of these two nodes is the same. Similarly consider an F such that it does not contain any conjunctions and it does contain a disjunction of the form $\phi_1 \vee \phi_2$. In this case the dependency graph has three successors to (s, F) ; they are labelled by $(s, F - \{\phi\} \cup \{\phi_1\})$, $(s, F - \{\phi\} \cup \{\phi_2\})$ and $(s, F - \{\phi\} \cup \{\phi_1, \phi_2\})$. These four nodes capture Equation 1 in Lemma 4.5 which characterizes the measure of trees satisfying a disjunction.

We now come to the hardest case in the construction of the dependency graph. Consider a node (s, F) such that all formula in F are of the form $\langle a \rangle \phi$ or $[a]\phi$. In this case for each a -successor s' of s we now create a node $(s', \{ \phi \mid \langle a \rangle \phi \in F \text{ or } [a]\phi \in F \})$. Let n be the node (s, F) let $A_n = \{a \mid (n, a, n') \text{ in the dependency graph}\}$ and let X_n stand for measure denoted by node n . These edges capture the constraint:

$$X_n = \prod_{a \in A_n} \sum_{(n, a, s') \in E} (P(s, a, s') \cdot X_{(s', F')})$$

This equation implicitly considers all possible deterministic trees that emanate from the state s . Clearly it gives rise to non-linear equations which need to be solved in calculating $m_{s_0}(\Theta_L(\psi))$. Once these equations are solved perhaps by using tools such as Maple we can easily check whether $\mathbb{E}_{\geq p} \psi$ holds at state s_0 .

7 Concluding Remarks

We have presented a uniform framework for defining temporal logics on reactive probabilistic transition systems. Our approach is based on using the modal μ -calculus to define measurable sets of observations of such systems. We have showed that our logic is expressive enough to encode two different existing temporal logics and we have also demonstrated that it may be used to rectify an infelicity in a third. A model-checking procedure for the logic was also presented.

An important issue for future work is that of applying our logic to more general transition systems (for example the transition systems of [14]) and establishing its relation to probabilistic automata [13]. Such an extension would allow us to deal with probabilistic models that are closed under the composition operator Γ a property lacking in our probabilistic labelled transition system.

References

- [1] J.C.M. Baeten and W.P. Weijland. *Process Algebra* volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press/Cambridge/England/1990.
- [2] G. Bhat and R. Cleaveland. Efficient model checking via the equational μ -calculus. In *Eleventh Annual Symposium on Logic in Computer Science (LICS '96)*. IEEE/Computer Society Press/July 1996.
- [3] E. Allen Emerson and Chin-Laung Lei. Efficient model-checking in fragments of the propositional μ -calculus. In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16-18 June 1986* pages 267-278. IEEE Computer Society Press/Los Alamitos/CA/1986.
- [4] E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In J. de Bakker and J. van Leeuwen/Editors/*Automata, Languages and Programming (ICALP '81)* volume 85 of *Lecture Notes in Computer Science*/Utrecht/July 1981. Springer-Verlag.
- [5] J.F. Groote and M. Rem/Editors. *Special issue of Science of Computer Programming*. Elsevier/North-Holland/To appear.
- [6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall/1985.
- [7] Michael Huth and Marta Kwiatkowska. Quantitative analysis and model checking. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science* pages 111-122/Warsaw/Poland/29 June-2 July 1997. IEEE Computer Society Press.
- [8] J. G. Kemeny/J. L. Snell/and A. W. Knapp. *Denumerable Markov Chains*. Van Nostrand/New Jersey/1966.
- [9] D. Kozen. Results on the propositional μ -calculus. *Theoretical Computer Science* 27(1):333-354/1983.
- [10] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation* 94/1991.
- [11] R. Cleaveland P. Iyer M. Narasimha. Probabilistic Model-Checking via Modal μ -calculus. In *Proc of FOS-ACS'99*.
- [12] R. Milner. *Communication and Concurrency*. PHI Series in Computer Science. Prentice Hall/1989.
- [13] Azaria Paz. *Introduction to Probabilistic Automata*. Academic Press/New York/1971.
- [14] R. Segala. A compositional trace-based semantics for probabilistic automata. In *CONCUR95* pages 324-338/1995.
- [15] Rob van Glabbeek/Scott A. Smolka/Bernhard Steffen/and Chris M. N. Tofts. Reactive/generative/and stratified models of probabilistic processes. In *Proc. of Fifth Annual IEEE Symposium on Logic in Computer Science* pages 130-141. IEEE Computer Society Press/June 1990.

Deductive Model Checking and Abstraction

Zohar Manna, Henny B. Sipma, and Tomás E. Uribe
Computer Science Department
Stanford University
Stanford, CA. 94305-9045
manna@cs.stanford.edu
(650) 723-4364

Abstract

We show how Deductive Model Checking, a method that combines deductive and algorithmic verification of temporal properties of reactive systems, can be understood as interactively specifying a finite-state *extended abstraction*. The transformations of DMC correspond to *refinement* operations, which construct an abstract finite-state system, and *model checking* operations, which check that it satisfies an abstract temporal property. The refinement steps make the overapproximated abstract transition relation smaller, and the underapproximated one larger. They also add fairness constraints to the abstract system, by computing abstract bounds on the enabling conditions of fair transitions, and eliminate unfeasible abstract loops by using well-founded orders.

1 Introduction

Methods for the verification of temporal properties of reactive systems have traditionally been classified as *deductive* or *algorithmic*. Deductive methods apply *verification rules*, which reduce the validity of temporal properties to that of *verification conditions*. Algorithmic methods, also known as *model checking*, prove temporal properties by exhaustively exploring the state-space of the system.

While algorithmic methods are automatic and can produce counterexamples when the property fails, they are usually restricted to finite-state systems, or specialized classes of infinite-state ones. On the other hand, deductive methods can verify general infinite-state systems, but require user interaction and do not produce counterexamples.

Deductive-algorithmic methods combine the two approaches. Since the problem of verifying general infinite-state systems is undecidable, they cannot be guaranteed to succeed or produce counterexamples; however, they may facilitate the verification task by allowing automatic methods to perform most of the combinatorial exploration, letting the user focus on higher-level aspects of the proof.

Abstraction presents a general approach to the combination of deductive and algorithmic methods: instead of proving properties of a given *concrete* system S , a simpler *abstract* system S^A is constructed, which can be model checked. The correctness of the abstraction is justified using deductive means, and guarantees that if a property holds for S^A , then a corresponding property holds for S . However, the converse is often not the case: if a property fails for S^A , it may or may not hold for S . The abstraction must then be *refined* until the property can be proved or disproved.

In this paper, we describe how the deductive-algorithmic method of Deductive Model Checking (DMC) [15, 14] does precisely this: it interactively constructs and refines an abstraction that can be model checked.

As we will see, DMC constructs the abstract system in a top-down, hierarchical manner, interleaving the model checking and the refinement operations. This can lead to space and time savings, since not even the full abstract state-space needs to be explored.

2 Preliminaries

2.1 Concrete Representation: Fair Transition Systems

Fair transition systems [13] are a convenient formalism for describing reactive systems. The representation relies on an *assertion language* to represent sets of states, usually based on first-order logic. A *state* of a fair transition system is a type-consistent interpretation of all its system variables V . The set of all states is called the *state space*, written Σ . An *assertion* is a first-order formula whose free variables are a subset of V ; it represents the set of states that satisfy it.

Definition 2.1 (Fair Transition System (FTS)) A fair transition system $S : \langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ is given by a set of system variables V , an initial condition Θ , expressed as an assertion over V , a set of transitions $\mathcal{T}$, relations on states, represented by assertions ρ over V and V' , where V' denotes the values of the variables in the next state, a set of just transitions $\mathcal{J} \subseteq \mathcal{T}$, and a set of compassionate transitions $\mathcal{C} \subseteq \mathcal{T}$.

An infinite sequence of states $\sigma : s_0, s_1, \dots$ is called a *computation* of S if (1) s_0 satisfies the initial condition, written $s_0 \models \Theta$, (2) for every $j \geq 0$, there exists a transition $\tau \in \mathcal{T}$ such that $\langle s_j, s_{j+1} \rangle \models \rho_\tau$, (3) it is not the case that a just transition is continuously enabled beyond some point, but never taken, and (4) every compassionate transition that is infinitely often enabled is infinitely often taken, where we say that a transition τ is enabled on a state s if there exists a state s' such that $\langle s, s' \rangle \subseteq \tau$, and a transition τ is taken at position j if $\langle s_j, s_{j+1} \rangle \subseteq \tau$.

2.2 Specification: Formula Automata

A temporal property can be specified by a *formula automaton*, or ω -automaton. Several types of finite-state ω -automata exist; here we will use the type known as Müller automaton.

Definition 2.2 (Müller automaton) A Müller automaton $\mathcal{M} : \langle N, N_0, E, \mu, \mathcal{F} \rangle$ over a domain Σ is a directed graph, consisting of a finite set of nodes N , a subset, N_0 , of which are initial, a set of edges E , a node labeling function μ that assigns to each node a subset of Σ and an acceptance condition $\mathcal{F} \subseteq 2^N$ given by a set of subsets of nodes.

An infinite sequence of states $\sigma : s_0, s_1, \dots$ is a *run* of an automaton $\mathcal{M}$ if there exists a path $\pi : n_0, n_1, \dots$ through $\mathcal{M}$ such that $n_0 \in N_0$, and for every $j \geq 0$, $s_j \in \mu(n_j)$. A run is a *model* of the automaton if it is accepted by the acceptance condition, that is, there exists a path π such that $\text{inf}(\pi) \in \mathcal{F}$, where $\text{inf}(\pi)$ is the set of nodes that appears infinitely often in π . Clearly the nodes that appear infinitely often have to be a strongly connected subgraph (SCS) of the automaton. Hence, only SCS's have to be considered in the acceptance condition.

2.3 Ranking Functions

Ranking functions are used to justify that loops cannot be traversed infinitely many times.

Definition 2.3 (Ranking functions) A binary relation $\succ$ over a domain $\mathcal{D}$ is a subset of $\mathcal{D} \times \mathcal{D}$, where we write $s_1 \succ s_2$ iff $\langle s_1, s_2 \rangle \in \succ$. A binary relation $\succ$ is well-founded over $\mathcal{D}$ if there are no infinite sequences of elements $e_1, \dots, e_n, \dots$ in $\mathcal{D}$ such that $e_1 \succ e_2 \succ \dots \succ e_n \succ \dots$. A ranking function δ is a mapping from system states into a well-founded domain $(\mathcal{D}, \succ)$. We write $x \succeq y$ iff $x \succ y$ or $x = y$.

2.4 Abstraction

We use a standard instance of abstraction [7], where the abstract system is given in terms of an *abstract domain* Σ_A , usually a complete lattice. For each abstract state in Σ_A , a *concretization function* γ gives the set of concrete states $\gamma(a)$ that a represents. An *abstraction function* α gives, for a set of concrete states, the smallest abstract state that includes it. The abstract domain we use is particularly simple, but has proved useful for generating and proving invariants [9, 1] and general temporal properties [6].

Definition 2.4 (Assertion-based abstraction) Given a finite set of assertions B , the assertion-based abstract domain with basis B has the complete boolean algebra $BA(B)$ (using $\wedge^A, \vee^A, \neg^A$) as its abstract domain Σ_A , where $\gamma(f) \stackrel{\text{def}}{=} \{s \in \Sigma \mid s \models f\}$, and $\alpha(S) \stackrel{\text{def}}{=} \bigwedge^A \{s^A \in BA(B) \mid S \subseteq \gamma(s^A)\}$.

That is, the concretization $\gamma(f)$ of an assertion f is simply the set of concrete states that satisfies it. The abstraction $\alpha(S)$ of a set of states S is the smallest point in the abstract lattice whose concretization includes all the elements of S .

If s^A is an abstract assertion (or, equivalently, an abstract state), then $\gamma(s^A)$ is characterized by the concrete assertion obtained from s^A by replacing $\wedge^A, \vee^A$ and $\neg^A$ by $\wedge, \vee$ and $\neg$. The boolean variables in s^A , which are elements of B , appear as corresponding subformulas in $\gamma(s^A)$. That is, γ is a *boolean homomorphism* between the two assertion languages.

If Σ_A is a correct abstraction of S and we can establish that $\Sigma_A \models \varphi^A$, then we will know that $\Sigma \models \gamma(\varphi)$, where $\gamma(\varphi)$ is obtained by replacing each assertion in φ^A by its concretization. These abstractions are *weakly preserving*: if Σ_A does not satisfy φ^A , we cannot conclude that Σ does not satisfy φ .

For an abstract binary relation τ^A , we define

$$\gamma(\tau^A) \stackrel{\text{def}}{=} \{(s_1, s_2) \mid s_1 \in \gamma(a_1) \wedge s_2 \in \gamma(a_2) \text{ for some } (a_1, a_2) \in \tau^A\}.$$

2.5 Abstract Representation: Extended Finite-State Abstractions

Following [16], our representation for abstract systems has the following components:

1. An over-approximated initial condition Θ_A , where $\Theta \subseteq \gamma(\Theta_A)$.
2. A set of abstract transitions $\{\tau_1^A, \dots, \tau_n^A\}$ that *over-approximate* the concrete ones, in the sense that $\tau_i \subseteq \gamma(\tau_i^A)$. These are known as the “free,” “liberal,” or “ $\exists\exists$ ” abstract transition relations [5, 8], and are used to prove universal temporal properties for the abstract system that can then be transferred to the concrete one.
3. A set of *constrained* abstract transition relations $\{\tau_{\forall\exists 1}^A, \dots, \tau_{\forall\exists n}^A\}$, where $\tau_{\forall\exists}^A(a_1, a_2)$ holds if *for all* $s_1 \in \gamma(a_1)$ *there exists* $s_2 \in \gamma(a_2)$ such that $\tau(s_1, s_2)$ holds. These are also known as the “conservative” or “ $\forall\exists$ ” abstract transition relations, and are used to prove existential temporal properties.
4. A *fairness table* that includes, for each fair transition τ_i^A , an abstract lower bound $\text{enabled}^-(\tau)$ on the enabling condition of τ_i^A , that is,

$$\gamma(\text{enabled}^-(\tau_i)) \subseteq \text{enabled}(\tau_i).$$

5. A *termination table*, which relates well-foundedness of relations at the concrete level with relations over the abstract system. Each row of the table contains:

- (a) A pair of ranking functions $\langle \delta_i, \delta_j \rangle_r$ over the concrete set of variables $\mathcal{V}$, over a well-founded domain $\mathcal{D}_r$.
- (b) A *precondition* pre_r and a *postcondition* post_r , which are *abstract assertions*, describing sets of abstract states.
- (c) A *result*, which is either $\prec$ or $\preceq$.

The verification conditions that justify the correctness of a termination table row r with result $\prec$ (resp. $\preceq$) are:

$$\gamma(\text{pre}_r^A)(\mathcal{V}) \wedge \tau_r(\mathcal{V}, \mathcal{V}') \wedge \gamma(\text{post}_r^A)(\mathcal{V}') \rightarrow \delta_{r,1}(\mathcal{V}) \prec (\text{resp. } \preceq) \delta_{r,2}(\mathcal{V}')$$

and

$$(\gamma(\text{pre}_r^A)(\mathcal{V}) \vee \gamma(\text{post}_r^A)(\mathcal{V})) \rightarrow (\delta_{r,1}(\mathcal{V}) \in \mathcal{D}_r \wedge \delta_{r,2}(\mathcal{V}) \in \mathcal{D}_r).$$

That is, whenever τ_r is taken from a state that pre_r^A represents, to reach a state that post_r^A represents, the well-founded order given by $\langle \delta_{r,1}, \delta_{r,2} \rangle$ decreases ($\prec$) or is not increased ($\preceq$).

3 Deductive Model Checking

Deductive model checking (DMC) [15] is a method for the interactive model checking of possibly infinite-state systems, generalizing the classic explicit-state model checking procedure for LTL [13]. The procedure itself is presented in detail in [15, 14], so here we focus on those aspects relevant to the abstraction point of view.

To verify that a system S satisfies a temporal specification φ , the classical procedure checks whether the $(S, \neg\varphi)$ *behavior graph*, that is, the product graph of the automaton representing $\neg\varphi$ and the transition graph, admits any counterexample computations. The DMC procedure starts with a skeleton of the behavior graph and progressively refines and transforms this graph until a counterexample is found or it is demonstrated that such a counterexample cannot exist. This graph is called the *falsification diagram* for S and φ . As their name suggests, falsification diagrams are dual to Verification Diagrams [12, 4]. Instead of showing that every computation of S can stay within an accepting SCS, as in the verification diagram case, we now show that every computation of S in the diagram, if any, *must* end in a non-accepting SCS.

Definition 3.1 (Falsification diagram) *Given an FTS S and a temporal property φ , a falsification diagram for S and φ is a directed graph $\mathcal{G} : \langle N, N_0, E, \mu, \eta, \kappa, \mathcal{F} \rangle$ consisting of an automaton $\langle N, N_0, E, \mu, \mathcal{F} \rangle$ over the state space of S , with two additional edge labeling functions, η and κ , which both label edges with subsets of $\mathcal{T}$, the set of transitions of S .*

An infinite sequence of states σ is a *model* of a falsification diagram $\mathcal{G} : \langle N, N_0, E, \mu, \eta, \kappa, \mathcal{F} \rangle$ if it is a model of the underlying automaton of $\mathcal{G}$; the underlying automaton of $\mathcal{G}$ is $\langle N, N_0, E', \mu, \mathcal{F} \rangle$ where $E' \subseteq E$ only includes those edges e such that $\eta(e) \neq \emptyset$. An infinite sequence of states is a *constrained model* of a falsification diagram $\mathcal{G}$ if it is a model of the constrained underlying automaton of $\mathcal{G}$, where the constrained underlying automaton is $\langle N, N_0, E'', \mu, \mathcal{F} \rangle$, with $E'' \subseteq E$ the set of edges e such that $\kappa(e) \neq \emptyset$. We write $\mathcal{L}(\mathcal{G})$ to denote the set of all models of $\mathcal{G}$, and $\mathcal{L}_c(\mathcal{G})$ to denote the set of constrained models of $\mathcal{G}$.

Figure 1 shows an outline of the Deductive Model Checking (DMC) procedure. The procedure repeatedly applies one of a set of transformations to the falsification diagram, until a counterexample computation is found or it is clear that no such computation can exist. At any given point, the models of the falsification diagram represents a superset of all the computations of the system that may possibly violate the temporal property, and the constrained models are a subset of the computations that do violate the temporal property.

3.1 DMC as Abstraction Refinement

By *abstraction refinement*, we mean the process of improving the approximations represented by an abstract system. This includes making $\exists\exists$ transitions smaller and $\forall\exists$ ones larger, while retaining the soundness of the approximation. Refinement also includes obtaining better bounds on enabling conditions and adding new well-founded orders.

Deductive Model Checking can be understood as the process of *refining* an abstraction of S , while simultaneously model checking it. The DMC transformations may be classified into two groups: the first performs model checking tasks, while the second performs abstraction refinement, justified by verification conditions.

Model checking a falsification diagram consists of checking whether the diagram embeds any abstract models. Formally, given a falsification diagram $\mathcal{G}$, an infinite sequence of states is an abstract model of $\mathcal{G}$ if it is a model of the underlying automaton of $\mathcal{G}$ over the abstract statespace defined by the atomic assertions in the node labeling. The set of all abstract models is written $\mathcal{L}^A(\mathcal{G})$. Abstract constrained models are defined similarly.

Note that for every (concrete) model of $\mathcal{G}$ there exists an abstract model, but not necessarily vice versa, since nodes may be labeled with unsatisfiable assertions other than *false*, eliminating sequences of concrete states passing through these nodes, but not the corresponding abstract ones. On the other hand, the verification conditions associated with the constrained transition ensure that any target node of an edge labeled by a constrained transition is satisfiable, and thus for every abstract constrained model there exists a corresponding concrete one.

After each refinement step we have three mutually exclusive possibilities:

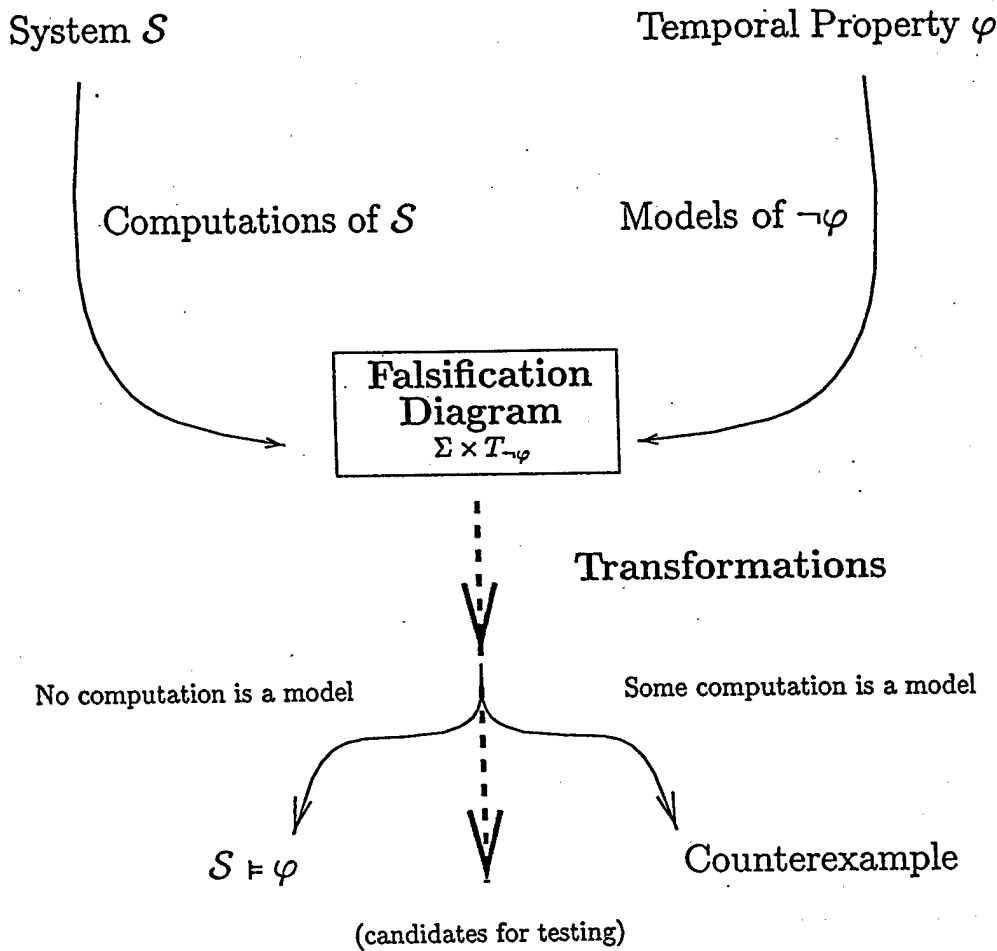


Figure 1: Outline of Deductive Model Checking (DMC)

1. $\mathcal{L}^A(\mathcal{G}) = \emptyset$. The abstract system can be successfully model checked, that is, $S^A \models \varphi^A$, where $\gamma(\varphi^A)$ in this case is equivalent to φ . Since S^A is a correct abstraction of S , this means that $S \models \varphi$.
2. $\mathcal{L}_c^A(\mathcal{G}) \neq \emptyset$. The abstract model checker can find, in the constrained $\forall\exists$ abstract relation for S^A , a counterexample to φ , in which case we know that $S \not\models \varphi$.
3. None of the above. The abstract model checker can only determine that $S^A \not\models \varphi^A$, finding an abstract counterexample, but cannot determine if a concrete counterexample exists. Thus, it may be the case that $S \models \varphi$, but S^A is too abstract to prove or disprove it. In this case, the abstraction must be *refined*.

A model checker specially tailored for this task is presented in [16]. A practical consequence is that a procedure similar to DMC can be implemented using a theorem prover and a model checker as black boxes, provided that the model checker gives feedback sufficient to select the next refinement transformation. If DMC succeeds, then a corresponding abstraction exists that can be model checked.

We should note that DMC provides an interactive, goal-oriented way of finding a suitable abstraction. Furthermore, it can selectively refine only those transitions and abstract states that are relevant to the property being proven. Thus, it can offer significant savings even for the finite-state case. The abstraction is computed “on-the-fly,” so that not even the entire *abstract* state-space has to be expanded.

The initial abstraction: The initial falsification diagram is the product of the $\neg\varphi$ automaton and the

abstract system

$$S_1^A : \langle \{b_1 : true^A\}, \theta : b_1, \mathcal{T} \rangle,$$

where each concrete transition τ_i is approximated by $\tau_i^A : true^A$. This is the coarsest abstraction of S , with basis $B : \{b_1 : true^A\}$, and only tautological (generally valid) temporal properties will hold for it. The initial $\forall\exists$ -approximations $\tau_{\forall\exists i}^A$ are $false^A$.

In DMC, edges in the product graph are labeled by sets of transitions. Initially, every edge is labeled by the set of all transitions.

3.1.1 Model Checking

- (model checking): The model checker checks whether $\mathcal{L}^A(\mathcal{G})$ is empty, in which case the property is proven, or whether $\mathcal{L}_c^A(\mathcal{G})$ is nonempty, in which case a counterexample has been demonstrated.

Note that in the abstraction framework, the only unsatisfiable nodes are those that are propositionally unsatisfiable. In DMC, the assertions from the automaton nodes and the states are combined and simplified. This corresponds to using theorem proving to decide relationships between the elements of the basis, ruling out unsatisfiable combinations.

3.1.2 Refinement: More Specific Transitions

We now consider two ways in which an abstraction can be refined. First, more information can be obtained about particular transitions by proving verification conditions that had not been considered previously, given the fixed basis of the atomic assertions that appear in the node labeling. In the second, new assertions are added to the basis. DMC provides transformations analogous to each of these:

DMC uses edge labels to keep track of which transitions *could* be taken at an edge. Transitions are removed from a label when we show that they cannot be taken at that edge:

- (remove edge label): If an edge from n_1 to n_2 is labeled with a transition τ , and the assertion

$$\mu(n_1)(\mathcal{V}) \wedge \mu(n_2)(\mathcal{V}') \wedge \tau(\mathcal{V}, \mathcal{V}')$$

is unsatisfiable, remove τ from the edge label.

This is equivalent to the *elimination method* of [1], and has the effect of conjoining $\mu(n_1) \rightarrow^A \neg^A \mu'(n_2)$ to the transition relation of τ^A . However, in DMC the refinement is local to a given edge: the transformation does not affect other edges where τ appears. We can optimize DMC by sharing the new information learned about τ^A throughout the falsification diagram. This has the effect of caching the proved verification conditions, reusing them whenever possible.

3.1.3 Refinement: Finer Abstract Domain

The second way to refine an abstract system is to choose a finer abstract domain, by introducing new basis elements. *Node splitting* transformations can be understood as doing precisely this:

- (binary split): Replace a node n by the two nodes n_1 and n_2 , labeled by

$$\mu(n_1) : \mu(n) \wedge \chi \quad \text{and} \quad \mu(n_2) : \mu(n) \wedge \neg\chi.$$

- (n -ary split): Replace a node n by the nodes $n_1, \dots, n_{j+1}$, labeled by

$$\begin{aligned} \mu(n_1) &: \mu(n) \wedge p_1 \\ &\vdots \\ \mu(n_j) &: \mu(n) \wedge p_j \\ \mu(n_{j+1}) &: \mu(n) \wedge \neg(p_1 \vee \dots \vee p_j) \end{aligned}$$

Removing transitions from edges after a split is equivalent to refining abstract transitions under a new extended basis. As described in [16], computing the abstraction that results from adding new basis elements or test points can be done without repeating the previous work.

3.2 DMC, Fairness and Well-founded Orders

In DMC terminology, a transition τ is *fully enabled* at a node n if $\mu(n) \subseteq \text{enabled}(\tau)$. We know that a transition cannot be taken at an SCS S if it has been removed from all the η edge labels in S . An SCS is then considered unfair if some just (resp. compassionate) transition is fully enabled at all (resp. some) nodes in S and is missing from all the edges in S .

DMC renders SCS's unaccepting according to the following criteria:

- They are unfair with respect to some transition τ , in the sense that no run can stay in the SCS without being unfair towards τ .
- They are well-founded, that is they have an edge that cannot be traversed infinitely many times.

Deductive Model Checking assigns ranking functions to the nodes of an SCS, to show that a computation cannot forever reside within the SCS, traversing all its edges and visiting all its nodes:

Definition 3.2 (Terminating edge) *An edge e_t in an SCS $S : \{n_1, \dots, n_k\}$ is terminating if there are ranking functions $\{\delta_1, \dots, \delta_k\}$ mapping states into a well-founded domain $\mathcal{D}$ such that:*

1. *for every edge $e : (n_1, n_2)$ in S and every $\tau \in e$*

$$\mu(n_1) \wedge \tau \wedge \mu'(n_2) \rightarrow \delta_1(\mathcal{V}) \succeq \delta_2(\mathcal{V}')$$

and

2. *for every $\tau \in e_t$,*

$$\mu(n_1) \wedge \tau \wedge \mu'(n_2) \rightarrow \delta_1(\mathcal{V}) \succ \delta_2(\mathcal{V}') .$$

We say that an SCS S for which a *tail*(S)-computation, that is, a computation that eventually ends up in the SCS S , cannot exist is *terminating*. The DMC procedure proves the termination of SCS's by identifying terminating edges. Clearly, if e_t is terminating in S , then no *tail*(S)-computation can traverse e_t infinitely many times. Such edges are removed from consideration, yielding smaller SCS's, until all accepting SCS's can be shown to be unfair, unreachable or well-founded.

In some cases, the ranking function for a node depends on the route taken to reach that node. An *unfolding* transformation makes copies of SCS nodes and allows proving that an SCS cannot support a computation even when suitable ranking functions cannot be found for the original SCS. With this machinery in place, DMC is relatively complete, as proved in [15, 14]. As usual, this assumes that the assertion language is expressive enough and that a complete proof procedure for the required verification conditions is available.

Concretization: DMC does not require an explicit concretization step, since the assertions in $\neg\varphi$ are built into the initial falsification diagram. Thus, the abstract property φ^A corresponds exactly to the original concrete φ , that is, $\gamma(\varphi^A) = \varphi$.

3.3 Counterexamples in DMC

So far, we have only used the (standard) $\exists\exists$ -approximation. Deductive Model Checking can also be used to find counterexamples, by collecting additional information that corresponds to the generation of an abstraction that also preserves existential properties. This is done by adding transitions to the edgelabeling function κ , which corresponds to adding new edges to the constrained ($\forall\exists$) transition relation.

- **(addition of transition in DMC).** Given an edge $e = (n_1, n_2)$, add τ to $\kappa(e)$ if the following formula is valid:

$$\forall \mathcal{V}. (\mu(n_1) \rightarrow \exists \mathcal{V}'. (\tau(\mathcal{V}, \mathcal{V}') \wedge \mu'(n_2))) .$$

This adds $\mu(n_1) \wedge^A \mu'(n_2)$ as a disjunct to $\tau_{\forall\exists}^A$. The set of transitions contained in the κ labeling function in a falsification diagram describes the corresponding $\tau_{\forall\exists}^A$ abstract transitions.

Dually to the "fully enabled" requirement on fair transitions for standard DMC, the existence of counterexamples requires that fair transitions that are not taken be *fully disabled* at the given nodes, that is, $\mu(n) \subseteq \neg\text{enabled}(\tau)$. We can then define:

Definition 3.3 (Fully fair) A transition is fully taken at an SCS if it is contained in the κ of an edge in the SCS. An SCS S is fully just (resp. fully compassionate) if every just (resp. compassionate) transition is either fully taken in S or fully disabled at some node (resp. all nodes) in S .

A counterexample exists if there is a reachable, accepting, fully fair SCS S in the underlying constrained automaton; a $\text{tail}(S)$ -computation will not violate any fairness requirements. Note that all the assertions in the respective nodes should be satisfiable. This is the case if the initial condition is satisfiable and the $\forall\exists$ -approximation is sound.

4 Related Work

The *possibility diagrams* of [11] and the *non-Zenoness diagrams* of [14, 3] are verification diagrams that also specify a $\forall\exists$ transition relation, allowing existential properties to be proved. Such diagrams can describe $\forall\exists$ and $\exists\exists$ abstract relations in a common diagrammatic form.

An important issue in practice is the strength of the validity checker and theorem proving available, and how well they handle first-order quantifiers. Decision procedures, integrated with first-order logic, can help with this task [2].

Other top-down methods for model checking infinite-state systems have been proposed, which are related to DMC; see [16] for a survey of some of these.

References

- [1] BENSALAM, S., LAKHNECH, Y., AND OWRE, S. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [10], pp. 319–331.
- [2] BJØRNER, N. S. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Computer Science Department, Stanford University, Nov. 1998.
- [3] BJØRNER, N. S., MANNA, Z., SIPMA, H. B., AND URIBE, T. E. Deductive verification of real-time systems using STeP. Tech. rep., Computer Science Department, Stanford University, Jan. 1999. To appear in *Theoretical Computer Science*. Preliminary version appeared in *4th Intl. AMAST Workshop on Real-Time Systems*, vol. 1231 of *LNCS*, pages 22–43. Springer-Verlag, May 1997.
- [4] BROWNE, A., MANNA, Z., AND SIPMA, H. B. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science* (1995), vol. 1026 of *LNCS*, Springer-Verlag, pp. 484–498.
- [5] CLEAVELAND, R., IYER, P., AND YANKELEVICH, D. Optimality in abstractions of model checking. In *Static Analysis* (Sept. 1995), vol. 983 of *LNCS*, Springer-Verlag, pp. 51–63.
- [6] COLÓN, M. A., AND URIBE, T. E. Generating finite-state abstractions of reactive systems using decision procedures. In Hu and Vardi [10], pp. 293–304.
- [7] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. Princ. of Prog. Lang.* (1977), ACM Press, pp. 238–252.
- [8] DAMS, D. R. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, July 1996.
- [9] GRAF, S., AND SAIDI, H. Construction of abstract state graphs with PVS. In *Proc. 9th Intl. Conference on Computer Aided Verification* (June 1997), O. Grumberg, Ed., vol. 1254 of *LNCS*, Springer-Verlag, pp. 72–83.
- [10] HU, A. J., AND VARDI, M. Y., Eds. *Proc. 10th Intl. Conference on Computer Aided Verification* (June 1998), vol. 1427 of *LNCS*, Springer-Verlag.

- [11] KESTEN, Y., MANNA, Z., AND PNUELI, A. Verification of clocked and hybrid systems. In *Embedded Systems*, G. Rozenberg and F. W. Vaandrager, Eds. Springer, Heidelberg, 1998, pp. 4-73.
- [12] MANNA, Z., AND PNUELI, A. Temporal verification diagrams. In *Proc. International Symposium on Theoretical Aspects of Computer Software* (1994), M. Hagiya and J. C. Mitchell, Eds., vol. 789 of *LNCS*, Springer-Verlag, pp. 726-765.
- [13] MANNA, Z., AND PNUELI, A. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
- [14] SIPMA, H. B. *Diagram-based Verification of Discrete, Real-time and Hybrid Systems*. PhD thesis, Computer Science Department, Stanford University, Dec. 1998.
- [15] SIPMA, H. B., URIBE, T. E., AND MANNA, Z. Deductive model checking. To appear in *Formal Methods in System Design* (1999). Preliminary version appeared in *Proc. 8th Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*, Springer-Verlag, pp. 208-219, 1996.
- [16] URIBE, T. E. *Abstraction-based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Computer Science Department, Stanford University, Dec. 1998.

Formal Methods in Practice

Wolfgang Polak

Consultant

wp@pocs.com – (408) 799-9210

Abstract

Technology transfer from academic research to industrial practice is hampered by social, political and economic problems more than by technical issues. This paper describes one instance of successful technology transfer based on a special-purpose language and associated translation tool tailored to the customer's needs. The key lesson to be learned from this example is that mathematical formalisms must be transparent to the user. Formalisms can be effectively employed if they are represented by tools that fit into existing work processes.

It is suggested that the model of special-purpose, domain-specific languages and their translators are an important vehicle to transition advanced technology to practice. This approach enables domain experts to solve problems using familiar terminology. It enables engineers of all disciplines to utilize computers without becoming software engineers. In doing so we not only mitigate the chronic shortage of qualified software personnel but also simplify the problem of requirements analysis and specification.

Keywords: Domain-specific languages; Code Synthesis; Technology Transfer

1. THE PROBLEM

The ultimate purpose of software engineering and computer science is to produce better, cheaper software. In this context *software* refers to a running system. The production of high-level source code is a possible but not necessary intermediate step. *Better* encompasses all qualitative aspects such as correctness, efficiency and so on. *Cheaper* refers to the overall cost of a software system including production, deployment, and maintenance.

Theoretical problems such as models for component composition, better theorem proving technology, formalized requirements analysis and the like, are important elements of a solution. The question is how best to make them practical.

Software engineers, desperate for automation, often create ad-hoc solutions without any formal basis. For example, the need to structure and organize complex software systems has led to the creation and success of UML. The question is how best to put these tools on a rigorous formal basis.

There is an impressive list of projects that use formal methods [1]. Yet most of the examples cited required extensive hand-holding by researchers and do not represent successful theory in wide-spread use. Examples of formal methods in common use are more modest and include grammars, supported by parser generators, and finite state machines [7].

Why does computer science not have a larger impact on software engineering practices? Clearly, there are many theories that should be valuable and useful in practice and there are many practical tools that

There is a big communication gap between theoreticians and practitioners. For the theoretician programs are mathematical objects that never fail if we can just get their specification right and verify the code. For the practitioner formal methods use obscure notation, deal with toy examples, and will never scale. Software engineers are faced with daunting management, version control, and similar problems and must constantly make engineering tradeoffs to meet tight deadlines and market windows – computer scientists know little of that. Computer scientists create wonderful theories, concepts and abstractions – software engineers understand little of that.

Transitioning science to engineering is not just a technical problem but is mainly an educational, social, managerial problem.

Educational: Software engineers could make use of many theoretical results if they knew how to do so. But we don't speak the same language. The presentation of research results is geared toward peer review not towards technology transition.

Social: Software engineers are reluctant to take outside advice. After all they manage to build complex systems.

Who likes to be told that some of his expertise can be replaced by a program?

Managerial: Processes and procedures for software construction have evolved over many years and are firmly entrenched in organizations. Any change will be perceived as risky and is likely rejected.

This paper describes an example of successful technology transfer based on an intelligent translator for a domain-specific specification language and lessons learned. Translators for very high-level languages provide a vehicle for

making complex, formally-based tools accessible to the engineering community. Indeed, special-purpose languages suggest a new paradigm of software development by empowering engineers in other disciplines to describe (aka program) solutions to their computational and control problems.

2. AN EXAMPLE OF TECHNOLOGY TRANSFER

2.1. The Problems of Technology Transfer

For several years the author was involved in a research project at a major aerospace corporation. The project studied techniques for program synthesis, automatic code generation, very high-level languages, graphical design tools and similar topics. The goal was to simplify specification of software systems and to make code synthesis practical by working in a restricted domain.

As in most industrial research laboratories there was the pressure to show practical relevance of the work. To that end, the project developed a number of prototype tools that were considered practical and useful by academic standards (e.g.[3; 2; 8; 4; 5]).

But academic standards are not good enough to be accepted by those responsible for real products. Several attempts to transition some of the lab's technology to product divisions were met with universal rejection. There were several reasons for this rejection, most of them non-technical in nature.

- Academics tend to develop tools in the abstract, i.e., they solve an intellectually interesting problem without regard to actual applications. When scientists talk about concepts such as "completeness of decision procedures" or "expressiveness of languages," their value will not be apparent to decision makers. Technology must be sold by describing the concrete problems being solved, how much time is saved, and how quality is improved. The technology is irrelevant, it is its impact that matters.
- People in charge of software projects are extremely concerned about schedule risk. Even if a new tool promises great time savings, it will be rejected if there is even minimal risk that it might negatively impact the schedule. Large potential time savings are often not realistic due to a steep learning curve.
- Researchers tend to build tools in isolation without consideration of the environment and the work process of software production. Tools that require changes in an established software development process are difficult to sell.
- An important reason for rejection is the perceived and often real lack of maintenance and support for systems that come out of research labs.
- One frequent objection to the use of machine generated code was readability. From the academic point of view, machine generated Ada code is no different than compiler generated assembly code. But the programmer in the field will be skeptical of the new technology and will want to inspect and understand the code. As a consequence significant effort was spent on generating human readable, commented code.

2.2. A Breakthrough

In early 1995, the company was preparing a proposal for a new NASA satellite program. To justify a low project cost an experiment was proposed that would demonstrate and measure the cost reduction possible through automatic code generation.

We were given an existing satellite software system that was operational in a simulator environment. The task was to generate from specifications one key module to achieve a different functionality. The generated code was to be tested and validated in the existing simulation.

After many failed attempts to introduce our technology into the product divisions we had finally generated some visibility and interest. There were a few major problems though. None of the lab's researchers had any experience with the satellite domain; we could not even understand the new requirements. We had no domain-specific specification language and no idea what one should look like. And we were only given four weeks to complete the experiment. The task was close to impossible. A cynic might think that we were deliberately setup for failure. More likely, the problem was of our own making since we had created misconceptions and wrong expectations in our earlier attempts to "sell" our technology.

After some fight, we convinced management to allocate a full-time aerospace engineer to the project. He was our domain expert and brought the specification language. As it turned out, aerospace engineers specify and test their control laws in MatLab¹. These MatLab specifications with some additional information became the input to our new tool.

¹MatLab and all other product and company names mentioned in this document are used for identification purposes only, and may be trademarks of their respective owners.

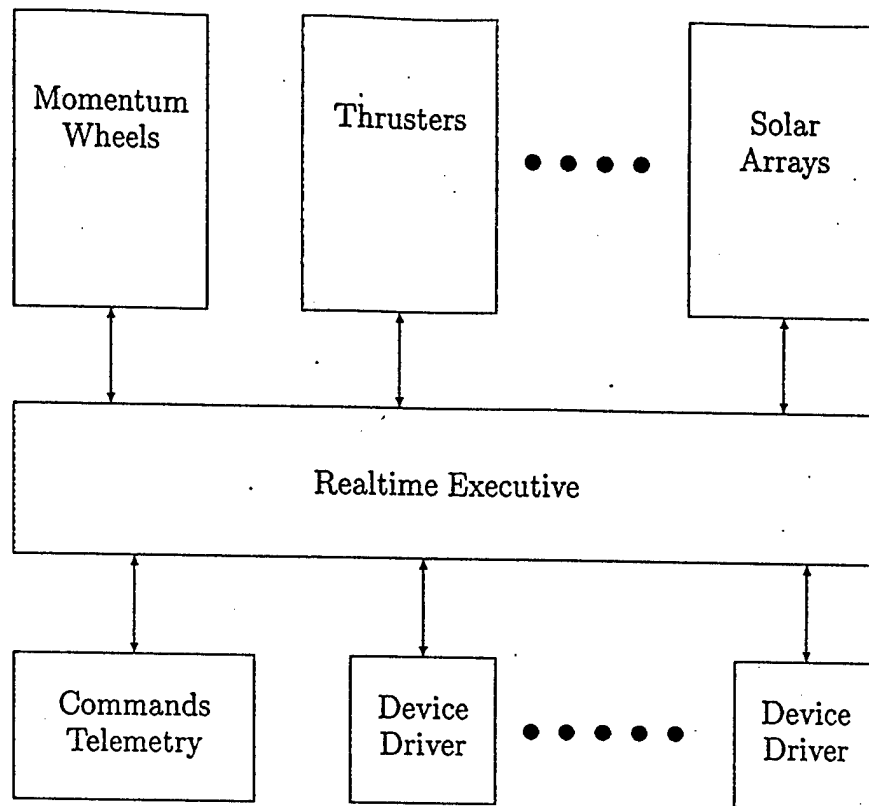


Fig. 1. Satellite software architecture with multiple functional modules that are connected to the realtime executive through standard interfaces.

Using extensive parsing, pretty printing, and tree manipulation tools that the project had developed over the years, we managed to build a prototype system that generated usable code – at least for one example. The experiment was successful and the data gathered was used in the proposal. Ironically, the proposal was not successful: its cost did not fit within the parameters considered reasonable by NASA and it was rejected as unrealistically cheap.

3. SUCCESSFUL AUTOMATIC CODE GENERATION

Even though the satellite proposal was not successful, the experiment demonstrated the utility of our approach and gave the lab some credibility. The aerospace engineer that participated in the experiment became a very strong advocate for the technology. By necessity (e.g., lack of time), we had created a solution that was simple and fit into the existing development process with minimal impact. As a result the initial crude prototype was further developed into a usable system, the Flight Code Generator (FCG), that is actively used on several programs. The current version of the system employs dataflow analysis, various code optimization techniques, type inference, and analysis of finite state machines.

FCG is successful because it (i) is specialized to a narrow domain, (ii) generates code that fits into an existing architecture, and (iii) fits into an established development process. The following is a brief description of these technical aspects of FCG.

3.1. Building Satellite Control Systems

Figure 1 shows a reusable software architecture for satellite control systems. The realtime executive provides an infrastructure that is independent of the particular system requirements and can be reused across multiple spacecraft. It connects spacecraft specific device drivers and functional modules. These modules perform such functions as rotating solar arrays, moving momentum wheels, determine position based on various sensors and so on. The code of each module is executed sequentially at an appropriate clock rate. For each clock cycle the module performs the appropriate computation which includes reading ground commands and sending telemetry information. Modules communicate by shared variables which require no synchronization if the reader and writer modules run at the same clock rate.

During the design process aerospace engineers (AE) develop the control laws for each functional module. Typically a single engineer works on a module. The control laws are coded and evaluated in MatLab to determine proper behavior.

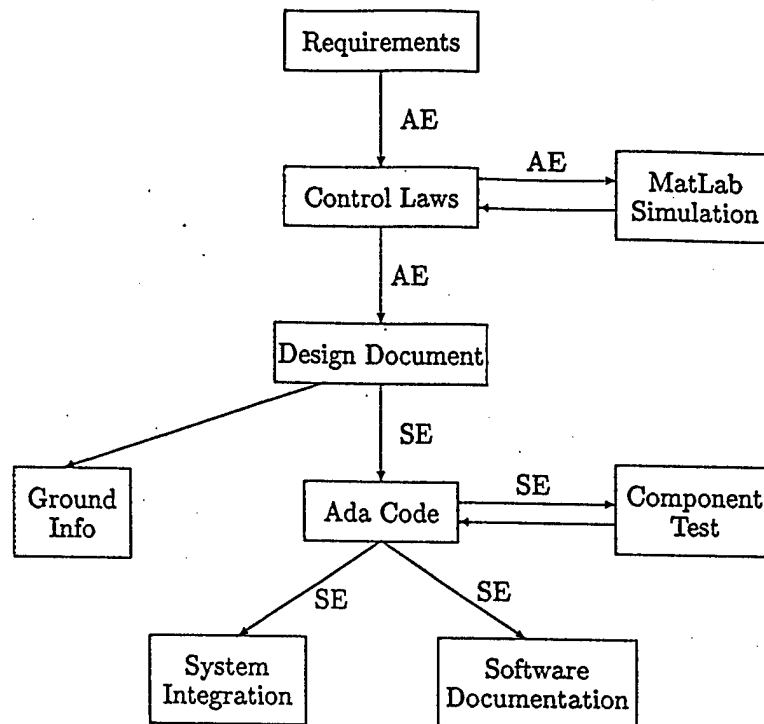


Fig. 2. Aerospace (AE) and software (SE) engineers cooperate to develop functional modules

The result of these tests are plots that show various responses to control inputs.

Figure 2 shows the development process for an individual module. A software engineer (SE) takes the design document produced by the aerospace engineer and develops appropriate Ada code. This code is unit-tested and later integrated into the system. Separate software documentation is produced for the hand-written Ada code. The design document is also used as a basis for developing ground software that needs to interpret telemetry information and generate commands.

3.2. Tool Support

It is apparent that the process of Figure 2 is inefficient and error prone. But it leaves plenty of room for automation and the experiment described in section 2.2 would not have possibly succeeded without the reusable architecture and the process being in place.

First, the process suggests a natural specification language: MatLab. While the MatLab source contains all necessary equations and formulas as well as test code to produce various plots, it does not contain information about the kind of telemetry to send, the commands and their parameters that are to be received, and how to respond to a particular command. Thus the specification language was defined as an extension to MatLab that includes the following additions:

- Optional type information can be added to determine precision of data and to select specific Ada types (e.g. the support infrastructure contains a 4-element float vector type as well as a quaternion type which are structurally equal but have different associated operations).
- Telemetry is specified by listing those variables whose values are to be included in the telemetry stream.
- Commands are defined by a name and possible parameters.
- A hierarchical finite state machine (essentially a textual version of state charts [6]) specifies the actions to be taken in response to a clock tick or a command.
- Special comments were added that can be included in generated Ada code and documentation.

In addition, it was necessary to mark certain inputs (e.g., test code that generates plots) so that it can be excluded from processing by FCG. All extensions were added to MatLab using special comment characters such that a source file of the extended language can still be processed by MatLab. The resulting language is ugly by any measure. But that problem was far outweighed by the benefits of having a single representation of the design. Engineers found surprising ways to make their specifications readable.

FCG is a batch tool written in Common Lisp that takes specifications written in the extended MatLab language

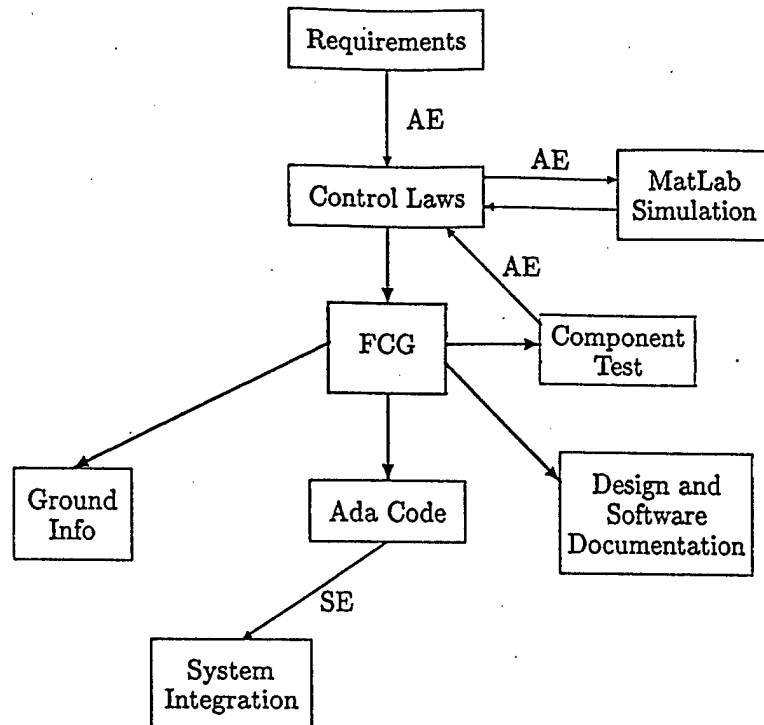


Fig. 3. FCG fits into the existing development process and eliminated virtually all manual handling of the Ada code for functional components.

and generates the following outputs (controlled by command line options)

1. Database records that describe telemetry and command information necessary for building ground software.
2. An Ada package that conforms to interfaces and conventions of the reusable architecture. While the code is commented and human readable it is ready for system integration and does not require human modifications.
3. A test environment that allows interactive or scripted unit testing of the generated Ada code. The test environment contains an interpreter that allows inspection and modification of all variables, calls to defined procedure, and the simulation of clock ticks and the arrival of commands. It also allows the generation of plots that can be compared with those generated by MatLab.
4. Documentation of both the design and implementation of the module. This information is based on the specifications, embedded comments, and decisions made by the Ada code generator

The new tool substantially simplifies the development process with only minimal additional work (see Figure 3). The aerospace engineer has to provide additional specifications in the MatLab source and is now performing unit tests of the generated Ada code. Any necessary code change is made in the MatLab source. Even with this additional work, the AE's job is simplified since the documentation requirements are reduced and the communication with the software engineer is eliminated.

3.3. A Recipe for Success

FCG is now used on three satellite systems. On one program FCG is being used both for the control and the payload software and almost half of the software is automatically generated. While this is significant, the system is not universally accepted throughout the corporation. Two problems dominate: The system lacks user support and maintenance. Many software designers refuse to work within the confines of a reusable architecture and insist on starting with a clean slate.

Why was FCG successful when much more elaborate earlier prototypes failed? Luck was an important part. The challenge experiment created the necessary visibility and convinced management and engineers of the value of the technology. Without the strong support of advocates from within the product division, insertion of new technology would not have been possible. Input from the user community is important. An internal advocate is ideal. Users that feel in control are very supportive. Interestingly, all support came from aerospace engineers whose jobs are more difficult with FCG. All resistance came from software engineers whose jobs are simplified by the tool.

Documentation is as important as code. Using a single source to generate code as well as documentation and

other artifacts ensures consistency and simplifies maintenance. Being able to generate custom database records and documentation was a major selling point.

A critical reason for success is minimizing risk. In the FCG approach it is always possible to revert to the old ways if problems should arise. Several features of the system helped to minimize risk:

1. The learning curve for the tool was very shallow. Initial use (e.g. unit testing) is possible using straight MatLab code.
2. The generated code is human-readable. If necessary, the code can be maintained by hand.
3. The tool fits into an existing development process. I.e., while some of the steps of the existing process are automated, none of the manual steps need to change in a significant way.
4. The system adapts to an existing architecture and its interfaces. No software changes are needed to accommodate machine generated code.

3.4. Commercial Tools

There are several commercial systems that generate code. But business reasons dictate that these systems are rather general purpose. Developing systems that generate custom code for a narrow domain is not commercially viable unless we can greatly simplify the construction and configuration of such system.

Integrated Systems offers MatrixX, a system for graphically specifying control systems and for generating code from such specifications. The product is much more mature and feature-rich than FCG but suffers from the lack of customization. The generated code cannot easily be integrated into the satellite architecture. MatrixX was actively considered but was perceived as much higher risk and more disruptive than FCG.

National Instruments' LabVIEW and BridgeVIEW are products for graphically designing data acquisition and signal processing applications.

Other examples of successful automatic code generators include parser generators and attribute grammar systems as well as numerous generators for graphic user interfaces.

4. FINAL THOUGHTS

Formal methods are a means, not an end. To become useful and accepted, computer science theory must be packaged and become invisible. Tool builders need to understand both the formalism and their end-users. Domain-specific tools provide a promising vehicle to deliver theory to practitioners.

Ever higher levels of specification provide increased opportunities for formal methods. Specifications based on constraints can use theorem provers to generate suitable code. Most domains tend to have design rules that can be checked using deductive or model-checking techniques. Domain-specific languages appear to be an effective delivery vehicle for formal methods. This, in turn, should reduce the cost and improve the quality of software.

Maybe domain-specific tools will eventually lead to a new software development paradigm, one where software technology empowers everyone to become a programmer in her field.

While the FCG experience provides only one data point, the existence of commercial tools (e.g. those cited above) is evidence that suggests that automatic code generation is accepted by practitioners. Domain engineers like to be in control rather than having to depend on software engineers.

Today software engineers are expected to play experts in all areas from human-computer interfaces to fluid dynamics to fly-by-wire systems. Software engineers cannot play all these roles and if they do, poor software is a necessity. Instead, software engineers should be tool builders. They are uniquely qualified to make computers accessible to other disciplines and to empower engineers in other fields to express their designs.

We have already seen how spreadsheet programs have made almost every computer user into a programmer. Obviously, not everyone is successful in programming their spreadsheets. But for disciplines where spreadsheets are in common use, their programming has already become part of the standard curriculum. In the long term, engineers in many disciplines will become programmers: domain specific programming will become part of the curriculum and standard practice in their discipline. Given the increasing proliferation of software, this development seems inevitable.

There is a good chance that such a development will also alleviate some of the problems of requirements analysis and capture. Requirements are often the interface between practitioners in different disciplines that speak different languages use different defaults and different common assumptions. If the requirements analyst and the programmer are experts in the same discipline there is much less change of miscommunication.

Acknowledgments

Eleanor Rieffel and James Baker provided valuable comments on earlier drafts of this paper. Discussions at the Monterey Workshop were very helpful and affected my thoughts on technology transition.

References

- [1] Edmund M. Clarke, Jeanette M. Wing, and et. al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4es):626-643, December 1996.
- [2] Henson Graves. Interactive design in LEAP. In *Proc. 91 AAAI workshop on Automating Software Design*, 1991.
- [3] Henson Graves. Lockheed environment for automatic programming. *IEEE Expert*, 7(6):15-25, December 1992.
- [4] Henson Graves, Joe Louie, and Tracy Mullen. A code synthesis experiment. In *7th Knowledge-Based Software Engineering Conference (KBSE-92)*. IEEE Computer Society Press, September 1992.
- [5] Henson Graves and Wolfgang Polak. Common intermediate design language. In *Hawaii International Conference on System Sciences*, January 1992.
- [6] D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8(3):231-274, 1987.
- [7] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE; a working environment for the development of complex reactive systems. In *Proceedings of the 10th International Conference on Software Engineering*, pages 396-406, Singapore, April 1988. IEEE Computer Society Press.
- [8] J. Williamson, P. Jensen, L. Ogata, and H Graves. Automatic programming technologies for avionics software (APTAS). In *Proceedings of the 9th Digital Avionics Systems Conference*, pages 101-106. IEEE, 1990.

Formalizing and Executing Message Sequence Charts via Timed Rewriting

Piotr Kosiuczenko, Martin Wirsing

Institut für Informatik, Ludwig-Maximilians-Universität, Oettingenstr. 67,
D-80538 Munich, Germany. E-mail: {kosiucze, wirsing}@informatik.uni-muenchen.de
Tel.: ++ 89 21782133, Fax: ++ 89 21782152

ABSTRACT

Message Sequence Charts (MSC) is a graphical trace language for describing and specifying the communication behaviour of distributed systems by means of message interchange. (Timed) Maude is a formal object-oriented specification language which combines algebraic specification techniques for describing complex data structures with (timed) term rewriting to deal with dynamic behaviour. In this paper we show first how to formalize MSC in Timed Maude. Then we give a translation of timed rewriting to untimed rewrite systems and use this translation to execute Message Sequence Charts with the Elan system, a powerful tool which combines Rewriting Logic with a language of rewriting strategies. We illustrate our approach with the bench mark example of a railroad crossing.

Keywords: Message Sequence Charts, formal methods, term rewriting, real-time systems, tool support.

1. INTRODUCTION

Currently used methods in software engineering use diagrammatic notations such as object diagrams, scenarios, Message Sequence Charts (MSC), or UML since their graphical representations facilitate reading, understanding, and the design of systems. Nevertheless these notations suffer from semantical problems, since their exact meaning is often unclear. Recently much effort has been made to overcome some of the semantical problems by applying formal techniques to methods used in practice of software engineering (e.g. [13]). This paper is another step in this direction. We show in this paper how to formalize Message Sequence Charts in the executable specification language Timed Maude and how to execute it with the Elan system.

Maude [2] is a formal object-oriented specification language which combines algebraic specification techniques for describing complex data structures with term rewriting to deal with dynamic behaviour. It is based on so called Rewriting Logic (RL) [9]. In Maude communication of system components is handled by interchanging messages. Timed Maude [7] is a real-time extension of Maude which adds a time view to the functional and the process view of Maude. It is based on Timed Rewriting Logic (TRL) in the same way as Maude is based on Rewriting Logic. TRL specifications with discrete time can be easily translated into untimed rewrite specifications in RL as we show in this paper.

MSC [4] is a graphical trace language for describing and specifying the communication behaviour of a distributed system by means of message interchange. It extends interaction diagrams by several constructs allowing to compose different diagrams. The language has been recommended as a standard by the International Telecommunication Union. A Maude like model can be used to provide a semantics for MSC [6]. In this paper we show how a basic MSC can be translated into Timed Maude following [5, 6]. We present also a formalization of two composition operators for horizontal and vertical composition [8] from MSC-96. The first operator allows one to define the sequential composition of specifications; the second one composes two systems working in parallel. Both operators are formalized using a syntactic substitution operator. We illustrate our approach with the bench mark railroad crossing example and present its formal specification. The specification is compositional since we specify smaller units and then glue them together using the horizontal and vertical composition operators.

Elan [1] is a powerful tool which combines Rewriting Logic with a language of rewriting strategies. It allows to describe and automatically execute rule based logical systems or processes. It provides also a modularisation mechanism for user defined modules to import other modules. We use Elan for executing MSC diagrams by translating their TRL semantics to Elan based on the interpretation of TRL in R. In the paper we show how to execute, analyze and improve the MSC specification of the railroad crossing using Elan.

Our approach has several advantages: it provides a direct translation of MSC's to timed rewrite specifications, it allows for compositional specifications by means of the composition operators, and makes available efficient tool support via powerful term rewriting systems like Elan or Maude.

The paper is organized as follows. In section 2 we give a short presentation of TRL and Timed Maude. In section 3 we show how TRL can be implemented by RL. In section 4 we present shortly the Timed Maude semantics of MSC. In section 5 we formalize vertical and horizontal composition of MSC's. In section 6 we demonstrate our approach by the railroad crossing example, present its Elan implementation and use it to execute the specification.

2. FORMAL BACKGROUND

Timed Rewriting Logic (TRL) [7] is an extension of Rewriting Logic (RL) [9] for describing real-time systems. It models time by archimedean monoids. Time evolves by executing rewrite rules. Each rule is labeled by a time stamp indicating the time needed for executing a rewrite step. The inference rules of TRL are similar to those of RL. The major difference to RL is the consideration of time stamps, the synchronous replacement rule and bounded (i.e. 0-time) reflexivity.

A *TRL-specification* is a pair of the form $(\Sigma(R_+), Ax)$, where $\Sigma(R_+)$ is a signature containing proper sorts and operation symbols and Ax is a set of equations and *timed rewrite rules* of the form: $t_1 \xrightarrow{a, r} t_2$, where a is a label denoting an *action* or a *system step*, r is a *time stamp* belonging to the underlying *arithmetical monoid* R_+^1 , and t_1, t_2 are Σ -terms coding *system states* (*configurations*). Informally, this means that the system evolves from state t_1 to t_2 by performing the step a in time r . TRL has the following deduction rules (plus rules for equational reasoning):

1. **Timed Transitivity (TT).** For each $t_1, t_2, t_3 \in T(\Sigma, X)$, $r_1, r_2 \in R_+$

$$\frac{t_1 \xrightarrow{a, r_1} t_2, \quad t_2 \xrightarrow{b, r_2} t_3}{t_1 \xrightarrow{a; b, r_1 + r_2} t_3}$$

2. **Synchronous Replacement (SR).** Let $\{x_{i1}, \dots, x_{ik}\} = FV(t_0) \cap FV(u_0)$ be the intersection of the free variables of t_0 and u_0 . For each $t_0, t_1, \dots, t_n, u_0, u_1, \dots, u_n \in T(\Sigma, X)$, $r \in R_+$

$$\frac{t_0 \xrightarrow{a, r} u_0, \quad t_1 \xrightarrow{a_{i1}, r} u_{i1}, \dots, t_k \xrightarrow{a_{ik}, r} u_{ik}}{t_0(t_1, \dots, t_n) \xrightarrow{a(a_{i1}, \dots, a_{i1}), r} u_0(u_1, \dots, u_n)}$$

3. **Timed Compatibility with = (TCE).** For each $t_1, t_2, u_1, u_2 \in T(\Sigma, X)$, $r_1, r_2 \in R_+$,

$$\frac{t_1 = u_1, \quad r_1 = r_2, \quad u_1 \xrightarrow{a, r_1} u_2, \quad u_2 = t_2}{t_1 \xrightarrow{a, r_2} t_2}$$

4. **0-time Reflexivity (0-R).** For each $t \in T(\Sigma, X)$

$$\frac{}{t \xrightarrow{t, 0} t}$$

1. Examples of arithmetical monoids are discrete time models like natural numbers as well as continuous time models like reals.

A term t is called *static* if it does not change over time, i.e. for all r , $t - t \ r \rightarrow t$ holds, and if $t - a \ r \rightarrow t'$ holds for some r , then $t = t'$. If a term t is not static then we call it *dynamic* and denote it by: $\text{op dynamic } t$ (see the applications section). Similarly, a sort s is *static* iff it contains static terms only, otherwise it is called *dynamic*.

Timed Maude is a formalism for specifying object-oriented distributed systems based on TRL. It borrows the object-oriented concepts of Maude. An (*object*) *class* is declared by an identifier and a list of attributes and their types. An *object* is represented by a term comprising a unique object name, an identifier for the class the object belongs to, and a set of attributes with their values. We will use capital letters for names and the corresponding small letters for objects being in certain states denoted by constants; e.g. the term $\text{tg} = \langle \text{TG} : \text{TrainGate} \mid \text{state} : \text{up} \rangle$ represents an object tg with name TG belonging to the class TrainGate , and the attribute state has value up . In the following we often omit the class and the name of the attribute and write simply $\langle \text{TG} \mid \text{up} \rangle$.

We assume that the set of possible *names* is a disjoint sum of a set of *object* names and a set of *gate* names. A *message* m is a term of the form (X, dt, Y) that consists of object or gate names X, Y of sort names and data dt of sort data carried by the message². X and Y can be understood as the sender and the receiver address, respectively. If X (Y , resp.) is a gate name, then it is to be understood, that the exact sender (receiver, resp.) address is unknown. Letters " g_i " and " $\bar{g}_i$ " will be used for gate names (g_i and $\bar{g}_i$ are dual names which can occur in "dual" specifications). Moreover, if X is a gate name, then we say that the message m is received from the *environment*; similarly, if Y is a gate name then m is sent to the *environment*. A *configuration* is a multiset (or bag) of messages and objects. Formally, configuration c is denoted by a term of the form:

$$m_1 \otimes \dots \otimes m_k \otimes o_1 \otimes \dots \otimes o_l$$

(shortly written: $m_1 \dots m_k o_1 \dots o_l$) where $\otimes$ is a binary function symbol denoting multiset union. A Timed Maude program makes computational progress by rewriting its state. A rewrite step has the form

$$m_1 \dots m_k o_1 \dots o_l - a \ r \rightarrow n_1 \dots n_p o_1' \dots o_w'$$

The rule says, that after receiving messages $m_1 \dots m_k$ the objects which occur on both sides of the rule change their states, objects which do not appear on the right hand side are deleted, objects which do not appear on the left hand side are created, and messages $n_1 \dots n_p$ are sent; all these happens in time r . Simple Maude allows only so called asynchronous rules, i.e. rules where $l = 1$.

3. INTERPRETING TRL IN RL

In this section we show how to translate timed TRL specifications to untimed rewrite systems written in RL [11]. In principle one could choose also any other term rewriting formalism, but RL is specially convenient due to its conceptual similarity to TRL. Our translation works for all linear TRL specifications with discrete time (and without equational axioms) and therefore can be applied to more general TRL specifications than the first translation of TRL to RL by Olveczky and Meseguer [10].

3.1 Definition

Let $Sp = (\Sigma(R_+), Ax)$ be an arbitrary TRL specification such that Ax contains rewrite rules only and each rewrite rule has time stamp 0 or 1. We define the RL interpretation $\text{Int}(Sp) = (\text{Int}(\Sigma(R_+)), \text{Int}(Ax))$ as follows:

The signature $\text{Int}(\Sigma(R_+))$ extends $\Sigma(R_+)$ with the following new (polymorphic) function symbols for every dynamic sort s :

$$\begin{array}{ll} \text{Go} : s \rightarrow s, & \text{D} : s \rightarrow s, \\ \text{clean} : s \rightarrow \text{Bool}, & \text{[]}(-) : \text{Time}, s \rightarrow s. \end{array}$$

The function Go (Go stands for *go*) is used to mark dynamic arguments where time must progress. The function D is used to mark where time has already progressed (D stands for *done*). These functions allow to synchronize progression of time in a term. The function clean ensures that a term (is ground and) does not contain Go or D . The function $\text{[]}(-)$ indicates the time available for executing a term.

2. In Maude a message is modelled by a pair of the form (dt, Y) where Y must be a name of an object, not a gate. There is no concept of gate in Maude.

The rewrite theory $\text{Int}(Ax)$ contains the following rules: The rules (i) allow us to simulate the 0-time rules on terms which do not contain Go or D; this is assured by the clean function. The cleanliness condition is needed to ensure that the RL rules modelling 0-time rewrites are not applied to a term part way through a time progression as this could lead potentially to a deadlock. The rules (ii) allow us to propagate time down through a term, in a sense. The meta-level term mapping Γ is used to describe how time progresses when the term is rewritten; it leaves static terms unchanged whereas dynamic terms receive new markings: Go if the time has to propagate down, and D if we are ready with a subterm. Rule (iii) serves to pull up the D function using another term mapping called Δ .

- (i) For each 0-time rule $t \rightarrow t' \in Ax$ we have the corresponding conditional rule:

$$t \rightarrow t' \text{ if } \text{clean}(t),$$
- (ii) For each rule $t \rightarrow t' \in Ax$ we have the corresponding conditional rule:

$$\text{Go}(t) \rightarrow \Gamma(t'),$$
- (iii) For each function symbol $f : s_1, \dots, s_n \rightarrow s \in \Sigma(R_+)$ we have a rule to pull up the D function:

$$f(\Delta(x_1), \dots, \Delta(x_n)) \rightarrow D(f(x_1, \dots, x_n))$$
- (iv) Finally, for each dynamic sort s and a variable x of sort s we have a rule to initiate the next time step:

$$[n+1](D(x)) \rightarrow [n](\text{Go}(x)).$$

The clean function is axiomatized by the following equations:

$$\begin{aligned} \text{clean}(\text{Go}(x)) &= \text{False}, & \text{clean}(D(x)) &= \text{False}, \\ \text{clean}(c) &= \text{True}, & \text{clean}(f(x_1, \dots, x_n)) &= \text{clean}(x_1) \& \dots \& \text{clean}(x_n), \end{aligned}$$

for each constant symbol $c \in \Sigma(R_+)$, and each n -ary function symbol $f \in \Sigma(R_+)$.

The term mapping $\Gamma : T(\Sigma(R_+), X) \rightarrow T(\text{Int}(\Sigma(R_+)), X)$ is defined as follows:

- (1) $\Gamma(t) = t$ if t is a member of a static sort;
- (2) $\Gamma(t) = D(t)$ if the term t is a member of a dynamic sort but contains no variables of dynamic sorts;
- (3) $\Gamma(t) = \text{Go}(x)$ if $t \equiv x$ and x is a dynamic variable ($\equiv$ is the syntactic identity);
- (4) $\Gamma(t) = f(\Gamma(t_1), \dots, \Gamma(t_n))$ if $t \equiv f(t_1, \dots, t_n)$, and t is of dynamic sort and contains variables of dynamic sorts.

The mapping $\Delta : X \rightarrow T(\text{Int}(\Sigma(R_+)), X)$ is defined by

$$\Delta(x) = x \text{ if } x \text{ is of a static sort, else } \Delta(x) = D(x).$$

In [11] it has been shown that the interpretation of TRL in RL is correct in the sense that a timed rewrite formula is deducible from a linear TRL specification if and only if its interpretation is deducible from the RL interpretation of the specification.

4. FORMALIZATION OF MSC

MSC is a trace language for description and specification of communication behaviour of system components and their environment by means of message interchange [4]. There is an interesting relation between Timed Maude and MSC, in particular, there exists a formal, Maude like semantics of MSC-96 [6].

Basically, a MSC-diagram describes a system behaviour in the following way [4]: the behaviour of an object (or instance) is represented by a vertical line defining a total ordering of its actions. Such a vertical line starts with an empty box and ends with a black box. If a message is being sent from one object to another, then this is indicated by a horizontal arrow directed from the sender to the receiver. Message passing is asynchronous; therefore it corresponds to two events: sending and receiving a message. Messages sent to (received from, resp.) the environment correspond to out (in, resp.) arrows labelled by gate names. One may specify the time a message deliverance will take by giving a real number under the horizontal arrow. The initial (final) states of an MSC are the states which occur directly after (before, resp.) an empty (black, resp.) box. There are only three kinds of atomic steps an MSC instance can execute: sending or receiving a message, creating or stopping another object, and a special action which changes only the state of an object. The first MSC diagram shown on figure 1

describes a train gate (named "TG"), which is initially in up position, and which moves down after receiving message m_{com} from message gate g .

Basic MSC can be formalized in Timed Maude as follows [5, 6]: For any instance O of a MSC diagram we introduce an object of appropriate class which contains an attribute called state, i.e. $\langle O : \text{Class} \mid \text{state} : s \rangle$. For simplicity, states will always be denoted by constants in this paper. Different object actions are separated by intermediate states. Object states may be described using (local) conditions denoted by hexagons (see below). Any message sent is split into a send and a receive part (indicated by "!" and "?" resp.). In general the underlying set L of *atomic steps* is of the form $\{?, m, !n, \text{start}, \text{stop}, \tau, ac_1, \dots, ac_n\}$ where "start" stands for creating an object, "stop" for deleting an object, " τ " for time elapse, and " $ac_1, \dots, ac_n$ " stand for special actions. Finally, we introduce a rewrite rule for any action as follows:

rl $\langle O \mid \text{state} : s \rangle - !m \ 0 \rightarrow \langle O \mid \text{state} : s' \rangle m$.
 **the object named O sends message m and changes its state from s to s' in 0-time

rl $m \langle O \mid \text{state} : s \rangle - ?m \ 0 \rightarrow \langle O \mid \text{state} : s' \rangle$.
 **the object named O receives message m and changes its state from s to s' in 0-time

rl $\text{start}(o) - \text{start } o \ 0 \rightarrow o$.
 **the object named O is created in 0-time

rl $\text{stop}(o) \ o - \text{stop } o \ 0 \rightarrow \epsilon^3$.
 **the object o is deleted in 0-time

rl $\langle O \mid \text{state} : s \rangle - ac_i \ 0 \rightarrow \langle O \mid \text{state} : s' \rangle$.
 **the object named O performs the special action ac_i changing its state in 0-time

rl $t_{\text{timer}}(r_1 + r_2) - \tau \ r_1 \rightarrow t_{\text{timer}}(r_2)$.
 **the value of timer t_{timer} is decreased by r_1 in time r_1

In all but the last case we say that the object named O *performs* the action $a \in L$ (e.g. TG performs the action $?m_{com}$). Unless stated otherwise, we will consider actions of the form $t_1 \otimes \dots \otimes t_n \otimes a$, $a \in L$, and of the form $t_1(\tau) \otimes \dots \otimes t_n(\tau)$ only for $t_i \in T(\Sigma, X)$, i.e. actions performed by one object only or rules where all components perform time step τ . For simplicity, we will write " a " for the former and " τ " for the latter.

The semantics of MSC's time aspects is based on the following assumptions (cf [6]): All terms are static per default, unless stated otherwise. All actions $a \in L$, except of τ , take 0-time. Time constraints imposed on an object behaviour or a message deliverance are modeled by timers attached to object states or to messages, respectively. An object can spawn multiple timers running in parallel. An atomic state (i.e. without timers) can be declared either as static or as dynamic. A static state can last arbitrarily long, whereas a dynamic state is supposed to change in time (i.e. its duration is 0-time units). In this paper we assume that the equational axioms concern only data carried by messages and the multisum operator $\otimes$. Formally, we define:

4.1 Definition

A *MSC-specification* SP is a tuple of the form $(\Sigma(R_+), Ax, In, Fin)$, where

- $(\Sigma(R_+), Ax)$ is a TRL-specification⁴ such that the underlying set of atomic labels as well as rewrite rules and axioms forming Ax have the form described above;
- If a gate symbol g ($\bar{g}$, resp.) occurs in signature $\Sigma(R_+)$, then the dual symbol $\bar{g}$ (g , resp.) does not; moreover, we assume that for each gate symbol there is exactly one object which communicates through the gate.

3. ϵ denotes the empty configuration.

4. For the sake of simplicity we treat here (flat) TRL-specifications instead of Timed Maude specifications in general.

- In is a set of objects in *initial* states and Fin is a set of objects in *terminal* states, both kinds of states are atomic (i.e. denoted by constants).
- Objects having different names must have different states.

The *interface* of SP is given by the set of gate symbols occurring in $\Sigma(R_+)$. A configuration $c = o_{i1} \dots o_{im}$ is called *initial* (*final*, resp.), if $c = \text{In}$ ($c = \text{Fin}$, resp.).

4.2 Definition

Let SP be a MSC specification. A *partial trajectory* is a finite sequence of configurations, actions, and time values $c_0 a_1 r_1 c_1 \dots c_{n-1} a_n r_n c_n$ such that c_0 is the initial configuration and for $i = 0, \dots, n-1$ one of the following conditions holds:

- m is a message received from environment, $a_{i+1} = ?m$,
and $mc_i - ?m \quad r_{i+1} \rightarrow c_{i+1}$;
- m is a message sent to the environment, $a_{i+1} = !m$,
and $c_i - !m \quad r_{i+1} \rightarrow c_{i+1} m$;
- $c_i - a_{i+1} \quad r_{i+1} \rightarrow c_{i+1}$ in any other case.

A partial trajectory is a *trajectory* if in addition c_n is the final configuration. $a_1 r_1 a_2 r_2 \dots a_n r_n$ is a (*partial*) *trace* of SP iff there exists a (partial) trajectory tr of the form $c_0 a_1 r_1 c_1 \dots c_{n-1} a_n r_n c_n$.

Note that for any (partial) trajectory tr messages sent to or received from the environment are not part of the configurations c_i . In this way we abstract from the moment when messages are sent or received by the environment, because the communication is asynchronous and these messages are not a part of a system configuration. This definition of trajectories allows us to specify system behaviours in a compositional way.

5. COMPOSITION OPERATORS

In this section, we are going to formalize two MSC concepts in Timed Maude which we call vertical and horizontal composition (cf [5]). We define first the vertical composition (which corresponds to so called weak sequencing) by means of state identification [8]. Then we consider parallel composition of components called horizontal composition defined as a simple form of pushout using readdressing of messages [8]. Syntactically we define both constructs using a renaming operator. Vertical and horizontal composition differ in what is being renamed: in the first case we substitute state names for state names, in the second object names for gate names.

The renaming operator is defined as follows: Let SP be a MSC-specification and let $p_1, \dots, p_n, q_1, \dots, q_n$ be constant symbols such that the sort of q_i is identical with the sort of p_i . Then $\sigma = (\text{op } p_1 \rightarrow q_1, \dots, \text{op } p_n \rightarrow q_n)$ is a signature morphism renaming $p_1, \dots, p_n$ into $q_1, \dots, q_n$. The specification $Sp * \sigma$ is obtained from Sp by simultaneously substituting $q_1, \dots, q_n$ for $p_1, \dots, p_n$ in Sp . The formula ϕ^σ is obtained from ϕ by renaming $p_1, \dots, p_n$ into $q_1, \dots, q_n$. Similarly, if A is a set of axioms, then $A^\sigma =_{\text{df}} \{\phi^\sigma \mid \phi \in A\}$. Below, we will substitute only object names for gate names (both of sort Identifier) and state names for state names.

5.1 Vertical composition

In our semantics we define vertical composition of two MSC-specifications by means of state identification. Namely, the final states of the first specification can be identified with the initial states of the second one, provided that they correspond

5. Semantically, $SP * \sigma$ corresponds to the translation “translate SP via σ ” where the specification-building operator “translate” of usual algebraic specifications is extended to MSC-specifications in the obvious way [12].

to equally named objects. The initial set of the first specification yields the initial set of the resulting specification and the final set of the second specification yields the final set of the resulting specification. We assume, that the specifications share the same basic data structures (like multisets, numbers, or lists), but have disjoint sets of states.

Formally, let $Sp_1 = (\Sigma(R_+)_1, Ax_1, In_1, Fin_1)$, $Sp_2 = (\Sigma(R_+)_2, Ax_2, In_2, Fin_2)$ be two specifications such that $\Sigma(R_+)_1$ and $\Sigma(R_+)_2$ are identical except that the corresponding sets of constant symbols denoting states must be disjoint, and that Ax_1 and Ax_2 contain the same equations. Let $Fin_1 = \{o_1', \dots, o_n'\}$, $In_2 = \{o_1, \dots, o_n\}$ and suppose that $\{st_1', \dots, st_n'\}$, $\{st_1, \dots, st_n\}$ are the corresponding final and initial states of the objects named $O_1, \dots, O_n$, respectively (i.e. Fin_1 and In_2 describe states of the same objects). Then $\sigma =_{df} (op\ st_1' \rightarrow st_1, \dots, op\ st_n' \rightarrow st_n)$ is a signature morphism which identifies the final states of the first model with the initial states of the second model.

5.1.1 Definition

The specification

$$Sp_1 \angle Sp_2 =_{df} (\Sigma(R_+)_1 \cup \Sigma(R_+)_2, Ax_1 \cup Ax_2, In_1, Fin_2) * \sigma$$

is called *vertical composition* of Sp_1 and Sp_2 .

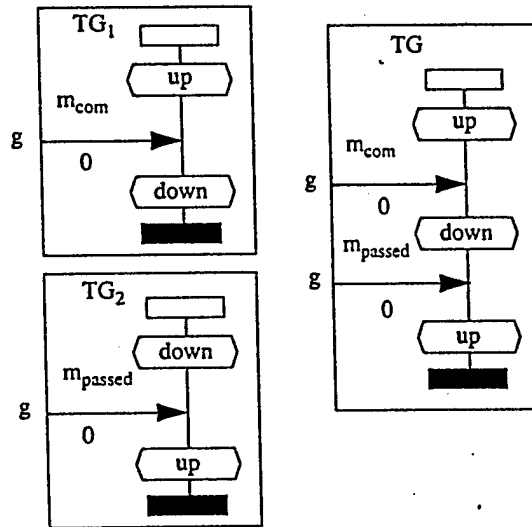


Figure 1. Vertical composition

Let us observe that σ has been derived from the initial and final states of Sp_1 and Sp_2 and therefore the resulting specification does not depend on σ . The set of initial states of $Sp_1 \angle Sp_2$ is In_1^σ , the set of final states is Fin_2 (which is equal to Fin_2^σ since σ renames only states of Sp_1).

5.1.2 Example

Let us consider a train gate TG of class TGate, that can be in two positions: up or down. Initially it is in up position, then it moves down immediately after receiving message m_{com} . Let us also consider a train gate which is initially in down position and moves up after receiving message m_{passed} . Let these messages have the form: $m_{com} = (\bar{g}, com, TG)$, $m_{passed} = (\bar{g}, passed, TG)$, where g is a gate. (Let us remind, that for a message gate g , " g " and " $\bar{g}$ " are dual names.) These two train gates can be specified as follows (we skip the class names):

Train gate specification $TGate_spec_1$ (see the top of figure 1):

op $up_1, down_1 : \rightarrow state.$

$up_1 : initial, down_1 : final.$

op dynamic $com : \rightarrow data.$

** this ensures that m_{com} is dynamic and therefore will be delivered immediately

rl $m_{com} \langle TG \mid up_1 \rangle - ? m_{com} 0 \rightarrow \langle TG \mid down_1 \rangle.$

** the train gate moves down immediately after receiving message m_{com}

Train gate specification $TGate_spec_2$ (see figure 1):

op $up_2, down_2 : \rightarrow state.$

$down_2 : initial, up_2 : final.$

op dynamic $passed : \rightarrow data.$

** this ensures that m_{passed} is dynamic and therefore will be delivered immediately

rl $m_{passed} \langle TG \mid down_2 \rangle - ? m_{passed} 0 \rightarrow \langle TG \mid up_2 \rangle.$

** the train gate moves up immediately after receiving message m_{passed}

We can compose these two specifications to a specification $TGate_spec$. It is defined as $TGate_spec_1 \angle TGate_spec_2$ using the renaming $\sigma = (op\ down_1 \rightarrow down_2)$. It specifies a train gate which can be in two positions: up, down, and which moves down immediately after receiving message m_{com} and moves up immediately after receiving message m_{passed} (see the right hand side of figure 1). The sets of initial and final states are as follows: $In = \{\langle TG \mid up_1 \rangle\}$, $Fin = \{\langle TG \mid up_2 \rangle\}$.

5.2 Horizontal composition

Using horizontal composition Message Sequence Charts system components can be composed for working in parallel. The difference between horizontal and vertical composition is that we glue the components not along states of equally named objects but along gates.

Let $Sp_1 = (\Sigma(R_+)_1, Ax_1, In_1, Fin_1)$ and $Sp_2 = (\Sigma(R_+)_2, Ax_2, In_2, Fin_2)$ and let $g_1, \dots, g_n$ be all those gate symbols such that g_i occurs in one of the specifications and $\bar{g}_i$ occurs in the other. Suppose, that the object named O_{ji} communicates through the gate g_i and O_{ki} through the gate $\bar{g}_i$, for $i = 1, \dots, n$ (O_{ji} and O_{ki} are uniquely determined according to the definition of MSC-specifications). Let the substitution σ have the form:

$$(op\ g_1 \rightarrow O_{j1}, \dots, op\ g_n \rightarrow O_{jn}, op\ \bar{g}_1 \rightarrow O_{k1}, \dots, op\ \bar{g}_n \rightarrow O_{kn}).$$

Moreover, we assume, that the specifications share the same basic data structures, i. e. they are defined by the same equations and that signatures are equal except of disjoint sets of state constants and sets of object names (names of object classes may overlap). We assume also that any gate different from $g_1, \dots, g_n$ occurs at most in one of the specifications Sp_1 or Sp_2 .

5.2.1 Definition

The specification

$$Sp_1 \oplus Sp_2 =_{df} (\Sigma(R_+)_1 \cup \Sigma(R_+)_2, Ax_1 \cup Ax_2, In_1 \cup In_2, Fin_1 \cup Fin_2) * \sigma$$

is called *horizontal composition* of Sp_1 and Sp_2 .

Specifications Sp_1 and Sp_2 are composed along gates $g_1, \dots, g_n$. Let us observe that the substitution σ is unique and that the interface of $Sp_1 \oplus Sp_2$ is given by those gates which occur in precisely one of the components.

6. APPLICATION: RAILRAD CROSSING

We illustrate our approach with the bench mark example of railroad crossing (e.g. [3]). First we present the informal (textual) specification, then its graphical and formal counterparts, and finally its translation to Elan [1].

Informal description

A railroad crossing consists of traingate TG and a train track T which has sensors attached to it. (see figure 2). Trains are moving along a track. When a sensor detects an incoming train on the track it sends immediately the message m_{com} to the train gate. When a sensor detects that a train has passed the crossing it sends immediately the message m_{passed} to the train gate. The train gate can be in two positions: up, down. It is initially in position "up", it moves down after receiving a message m_{com} , and moves up after receiving a message m_{passed} . We impose the following time constraints on the system behaviour:

- Trains are separated by a period of at least 8 minutes.
- A sensor detects the arrival of a train before the train reaches the train gate. It sends the message m_{com} to the train gate as soon as it detects a train arrival and the message m_{passed} as soon as the train has passed the crossing.
- The train gate moves immediately down (up, resp.) from position up (down, resp.) after receiving message m_{com} (m_{passed} , resp.).
- Every message is delivered immediately (in 0-time).

6.1 Formal design

An abstract design of this system can be given by a Message Sequence Chart which shows the message flow and the real-time constraints. The timings of a message deliverance are indicated by a time value below the message arrow. Setting a timer in an object (e.g. t_{far} , see figure 2) is indicated by a double triangle. Timers are set with certain values (t_{far} is set with value 8). Time-outs are indicated by arrows starting in the double triangles.

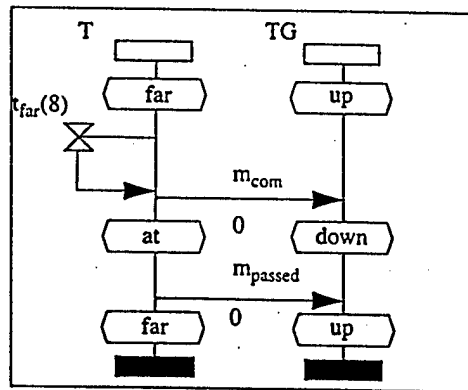


Figure 2. MSC of railroad crossing

A Message Sequence Chart showing the railroad crossing is given in figure 2. The train gate TG is modelled as in section 4. The behaviour of train and sensor is described by T. A train is initially in state far. A timer is set to 8 time units ensuring the 8 minutes distance between two trains. Sometime after the time-out of the timer the message m_{com} is sent changing the state of T to at (indicating that a train has arrived at the crossing). Later the message m_{passed} is sent and the state of T is set back to far (indicating that the train has left the crossing).

The corresponding Timed Maude specification is defined as horizontal composition of the specification *TGate_spec* of section 4 and a specification *Trains_spec* of train and sensor. These specifications are derived from the decomposition of the MSC for the railroad crossing (figure 2) into two parts T and TG, see figure 3. A train T (with class name Train) is an object

with an attribute storing information about the next train. The attribute of the train gate object TG (with class name "TGate") stores information about the gate position. Let the messages m_{com} and m_{passed} be of the form (T, com, g) , $(T, passed, g)$, respectively.

The train specification *Trains_spec* is as follows (see the MSC T on the left hand side of figure 3):

```

op fari, far', farf, at : → state ,
    fari : initial , farf : final .

** the states are static and can last arbitrarily long

op dynamic  com , passed : → data ,
    tfar : Time → Timer .

** messages mcom, mpassed will be delivered in 0-time because com, passed are declared dynamic
rl  <T | fari> → set tfar 0 → <T | tfar(8) ⊗ fari> .
** timer tfar is set and in 8 minutes a train may come
rl  <T | tfar(0) ⊗ fari> → tout tfar 0 → <T | far'> .
** 8 minutes elapsed, the timer times out, and the train may appear any time
rl  <T | far'> → !mcom 0 → <T | at> mcom .
** train detection immediately triggers sending message mcom immediately
rl  <T | at> → !mpassed 0 → <T | far'> mpassed .
** train passed and message mpassed is immediately sent

```

The formal specification *Spec* of the railroad crossing is defined as horizontal composition of *Trains_spec* and *TGate_spec*, i.e.:

$$Spec =_{df} Trains_spec \oplus TGate_spec.$$

The underlying substitution σ has the form $\sigma = \sigma_1 \sigma_2$ where $\sigma_1 = (op\ g \rightarrow S, op\ \bar{g} \rightarrow T)$ and $\sigma_2 = (op\ g \rightarrow TG, op\ \bar{g} \rightarrow S)$. Let us also observe, that the initial and final sets are as follows:

$$In = \{ \langle T | state : far_i \rangle, \langle TG | state : up_1 \rangle \},$$

$$Fin = \{ \langle T | state : far_f \rangle, \langle TG | state : up_2 \rangle \}.$$

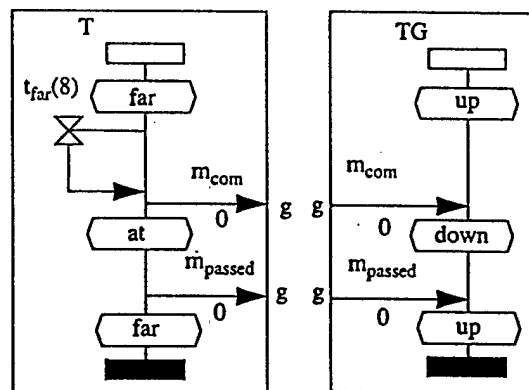


Figure 3. The components of the railroad crossing

6.2 Implementation

In this subsection we implement the above specification in Elan. The specification is composed of three modules: the module template containing the declaration of basic sorts and operations (we skip this module because of lack of space), the module *tGate_spec* specifying the train gate, and the module *trains_spec* specifying the whole railroad crossing. The module *trains_spec* imports the module *tGate_spec* which in turn imports template.

The module `tGate_spec` is a literal translation of *TGate_spec* according to definition 3.1.

```

module tGate_spec
  import global
    template ;
  end
  operators
    global
    up1 : state ; up2 : state ; down2 : state ;
    com : data ; passed : data ;
    TG : obn ; T : obn ;
  end
  rules for conf
    global
    [] m(T, com, TG) * <TG | up1> => <TG | down2> end
    [] m(T, passed, TG) * <TG | down2> => <TG | up2> end
  end
end

```

For the specification of *T* we follow the interpretation schema except in case (ii) where the distributivity of *Go* is restricted to configurations without pending messages and where time progress in a timer is modelled as time progress in a whole object. The first change ensures that all messages will be read before the time progresses, the second makes the specification more compact. Let us observe that every 0-time rule applies only to terms having no variables, therefore we do not need the cleanliness function.

```

module trains_spec
  import global tGate_spec ;
  end
  operators global
    fari : state ; far' : state ; farf : state ; at : state ;
    t_far( @ ) : ( int ) timer ;
  end
  rules for conf
    s, x, y : state ;
    c1, c2 : conf ;
    n : int ;
    global
    [] <T | fari> => <T | t_far(8) & fari> end
    [] <T | t_far(0) & fari> => <T | far'> end
    [] <T | far'> => <T | at> * m(T, com, TG) end
    [] <T | at> => <T | farf> * m(T, passed, TG) end
    [] D(c1) * D(c2) => D(c1 * c2) end
    [] Go(<TG | x> * <T | y>) => Go(<TG | x>) * Go(<T | y>) end
    [] Go(<TG | x>) => D(<TG | x>) end
    [] Go(<T | farf>) => D(<T | farf>) end
    [] Go(<T | t_far(n) & s>) => D(<T | t_far(n-1) & s>) if n > 0 end
    [] [n] D(cf) => [n-1] Go(cf) if n > 0 end
  end
end

```

The Elan implementation allows us to execute the specification. We have tested the railroad crossing specification systematically for inputs ranging from 0 to 15 minutes (of railroad crossing system time) starting from the initial configuration. For example, we have rewritten the term: $[15] D(\langle T \mid \text{fari} \rangle * \langle TG \mid \text{up1} \rangle)$. The execution time was 0.380 seconds. Elan provides a possibility to trace the rewriting. By analyzing the execution trace we have observed that the specification, as it stands, does not prohibit racing between messages m_{com} and m_{passed} , since the second message can be read before the first one. This is due to the fact that there are no time constraints on the train speed. The train may reach the gate in 0-time before m_{com} will be read. This will not happen if we ensure that the train moves with a bounded speed and therefore reaches the train gate after a specified time elapse; a requirement like this may be specified by another timer.

The Elan implementation provides also more sophisticated facilities for executing specifications using the built-in strategy language one may specify complex search algorithms such as depth first search, or constrain the rewriting. In the future work we are going to use it for automatically checking simple temporal properties of the specified systems.

REFERENCES

- [1] P. Borovansky, C. Kirchner, H. Kirchner, P. -E. Moreau, C. Ringeissen. An Overview of ELAN. C. Kirchner, H. Kirchner (eds.): Proceedings of WRLA'98. Electronic Notes in TCS, 1998 (To appear).
- [2] P. M. Clavel, S. Ecker, P. Lincoln, J. Meseguer. Principles of Maude. In J. Meseguer (ed.): Rewriting Logic and its Application, Proc. of the First Int. Workshop, Electronic Notes in TCS, Vol. 4, 1996, pp 65-90.
- [3] A. Gabrielian. State Machines, temporal logic and algebraic data models. In T. Rus and C. Rattray (eds.): Theories and Experiences for Real-Time System Development, AMAST Series: Vol. 2, World Scientific, 1994, pp 239-263.
- [4] ITU. ITU-TS, Recommendation Z.120. Message Sequence Charts (MSC). ITU-TS, 1996, Geneva.
- [5] P. Kosiuczenko. Time in Message Sequence Charts: a Formal Approach. Proc. of EuroPar'97, LNCS 1300, 1997, pp 562-566.
- [6] P. Kosiuczenko. Toward a Formal Semantics of MSC-96: Inline Expressions. Technical Report Nr. 9705. Ins. für Informatik, Ludwig-Maximilians-Universität, München, 1997, 18 pages.
- [7] P. Kosiuczenko, M. Wirsing. Timed Rewriting Logic with an Application to Object-Based Specification. Science of Computer Programming. 28 (2-3), 1997, pp 225-246.
- [8] P. Kosiuczenko, M. Wirsing. Towards an Integration of Message Sequence Charts and Timed Maude. Proc. of IDPT Conference, Berlin, July 6-9, 1998.
- [9] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science 96, 1992, pp 158-200.
- [10] P. Olveczky, J. Meseguer. Specifying Real-Time Systems in Rewriting Logic. In J. Meseguer (ed.): Rewriting Logic and its Applications. Electronic Notes in TCS, 4. First International Workshop, Pacific Grove, C. A., September 1996.
- [11] J. Steggles, P. Kosiuczenko. A Formal Model for SDL Specification Based on Timed Rewriting Logic. Submitted for publication, November 1998.
- [12] M. Wirsing. Algebraic specification. In J. van Leeuwen (ed.): Handbook of Theoretical Computer Science, Amsterdam, North-Holland, 1990, pp 675-788.
- [13] M. Wirsing, A. Knapp. A Formal Approach to Object-Oriented Software Engineering. In J. Meseguer (ed.): Rewriting Logic and its Application, Proc. of the First Int. Workshop, Electronic Notes in TCS, Vol. 4, 1996, pp 321-359.

Real-Time Systems Development with MASS *

Vered Gafni^{1,3}

Yishai Feldman²

Amiram Yehudai¹

¹Computer Science Department, Tel-Aviv University, Tel-Aviv, Israel

²School of Computer and Media Sciences, The Interdisciplinary Center, Herzliya, Israel

³MBT, Israel Aircraft Industries, Yehud, Israel

Abstract

In this paper, we demonstrate the capability of MASS, a real-time design language, for large systems specification. The paper presents a hierarchical specification of an automatic cruise controller that evolves through stepwise refinement. In particular, we show modular design, the separation of the functional and reactive concerns, and the succinct and intuitive nature of specifications in MASS.

1 Introduction

A real-time system consists of a plant where dynamic processes take place, and a controller (an embedded computer) aimed at the stabilization of the on-going processes at a required state. The controller design is especially complex, as compared with non-real-time applications, due to the reactive aspect of the its operation. This aspect comprises the need to synchronize its computations with the occurrences of the plant events (indicated by sensor data) and to accomplish their executions within hard deadlines determined by the controlled process dynamics (typical applications handle a large number of asynchronous events, thus giving rise to many intricate scenarios). On the other hand, real-time systems are usually safety-critical, and therefore their correctness is of an essential importance.

MASS, the real-time design language employed in this paper, is based on a specification approach that handles the inherent complexity of real-time systems by completely separating the concerns of the functional and the reactive aspects in their specification. However, the underlying model formally relates these aspects in a way that enables comprehensive reasoning about the system behavior.

The key idea of the separation paradigm is to represent events and functions as different aspects of a single entity, called a *task*. In the functional view, a task denotes a computation that does not synchronize during execution with its environment (other tasks, or external systems). In the reactive view, a task is considered as a basic event that occurs at every termination of the task computation. Event expressions (constructed from basic events) are used to specify the activation of the tasks computations. Hence, due to the identification of events with task terminations, the specification of the computation of each task (given in the functional part) is completely independent of the reactive behavior in which it is employed.

MASS¹ employs a single specification construct, called a *reaction*, that expresses an activation requirement for a task's computation in response to events (namely, terminations of tasks' computations). For instance, the requirement "every occurrence of p triggers the computation associated with q that must terminate within 5 time units" is succinctly expressed in MASS by the reaction: $[p \Rightarrow q] \leq 5$. MASS also provides a unique mechanism that enables hierarchical specification of large systems by stepwise refinement

A MASS specification is made executable by a translation into a regular expression over signals denoting task activations and terminations. Operationally, we construct an automaton that operates synchronously

*Supported in part by grants from the Israel Science Foundation, the German-Israeli Foundation for Scientific Research and Development (GIF), and the Israel Ministry of Science.

¹MASS is acronym of 'Marionettes Activation Scheme Specification', a metaphor suggesting the separation of the activation mechanism from the activated puppets.

to monitor the execution of asynchronous computations. A compiler, a run-time system, and simulation package, constructed for MASS, enable its practical application to systems development. Indeed, numerous large-scale systems have been specified and implemented using MASS [5, 9].

MASS is part of a formal specification framework for real-time systems which also includes the real-time logic PLOT, and verification procedures that check the correctness of a design in MASS with respect to system requirements expressed in PLOT. In a nutshell, PLOT is an interval temporal logic [7] that allows the explicit expression of durations and timed occurrences of events. The logic is novel in its notion of causality, which is treated as a primitive semantic object. PLOT is decidable, and is associated with a deductive proof system. For the purpose of formal verification, a reaction is also interpreted by a PLOT formula that is consistent with the operational interpretation in the sense that it defines the same set of runs.

The rest of the paper is organized as follows. Section 2 provides an overview of MASS. Section 3 presents a worked-out example of a large system specification with MASS, and Section 4 surveys related work.

2 Overview of MASS

The specification unit in MASS, called an *act*, presents the reactive behavior of a real-time system in the following form (boldfaced tokens are reserved words, and the notation $t \dots$ means a finite list of terms of the type t).

```

Act name is
  Tasks
    system task, ...
    environment task, ...
  Reactions reaction ...
  TimeBase unit
End

```

The Tasks section presents the tasks that comprise the real-time system, classified into the environment and system types. Environment tasks represent plant activities that are observable by the controller, but are not under its control (recall that in the MASS model an activity is observed only by sensing the terminations of its executions). In contrast, the behavior of system tasks is fully controlled by the controller as specified by the reactions. The Reactions section specifies the activation requirements for the system tasks, and the Timebase section defines the sampling rate of the controller operation.

Tasks are declared in a functional notation of the form: $\text{task-id} : \text{Input} \rightarrow \text{Output}$ where task-id is a name, and Input and Output are finite domains. We use the term void to denote a singleton. However, declarations of the form $q : \text{void} \rightarrow \text{Output}$ and $q : \text{void} \rightarrow \text{void}$ are usually abbreviated to $q : \rightarrow \text{Out}$ and q , respectively. Henceforth, we assume all tasks are declared with a void input domain, as indeed it turns to be the case in the example presented in this paper.

Every task is associated with a set of basic events, $q = v$ where v ranges over the output domain (in case of a task $q : \text{void} \rightarrow \text{Output}$ the only basic event is also denoted by q). Each occurrence of an event $q = v$ denotes the termination of an execution of q that returned the output value v .

The events with respect to a given act are the basic events derived from the act tasks, the time events: startup, 0, 1, *, and every event expression of the form: $\neg\alpha$, $\alpha \vee \beta$, $\alpha ; \beta$, $\alpha \leadsto q$ where α, β are events, and q is a task.

The semantic domain for MASS is a real-time systems model represented by timed-state sequences. We assume that basic events (the terminations of task executions) are observable along a discrete time axis (modeled by $\mathbb{N}$). Every task q is associated with a set of events C_q considered its causes. A trace of q is a function $tr_q : \mathbb{N} \rightarrow 2^{(C_q \times \mathbb{N})}$. Each element $(\alpha, i) \in tr_q(t)$ indicates an execution of q that terminated at t and was activated due to an occurrence of the cause α at the time instant $t - i$ (a trace value $tr_q(t) = \emptyset$ means that no execution of q terminated at t).

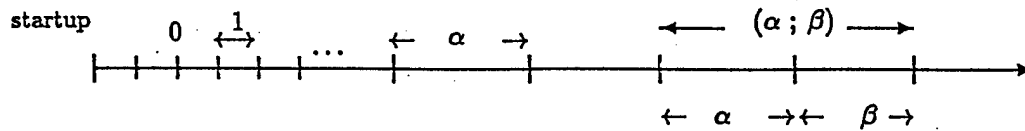


Figure 1: Illustration of MASS events

A trace represents a possible behavior of a task. A behavior of the entire system, called a *run*, consists of a trace for every task. Note that fixing the values of a trace as sets means that we allow concurrent executions of the task's computation, with the possibility that some of them terminate simultaneously (they are distinguished, however, by the causes and the activation times).

Events are interpreted over closed time intervals (including singletons $[n, n]$ representing time instants) with respect to a given run. Informally (see Fig. 1):

- A basic event occurs at every time instant t such that $tr_q(t) \neq \emptyset$.
- $\alpha \leadsto q$ occurs together with those occurrences of q that were caused by α (namely, at the time instants t such that there exists an element $(\alpha, i) \in tr_q(t)$).
- *startup* occurs only at the instant the system starts operating, 0 occurs at every time instant, 1 occurs at every unit interval $[n, n+1]$, and $*$ occurs on every time interval (specifying an arbitrary duration).
- The logical symbols $\neg, \vee$ denote negation, and disjunction, respectively.
- The symbol $;$ is the standard chop operator of interval temporal logic. The event $\alpha; \beta$ occurs at any interval composed of an occurrence of α immediately followed by an occurrence of β .

The standard temporal operator "eventually" is defined by $\Diamond \alpha \stackrel{\text{def}}{=} (*; \alpha; *)$, its dual "always" by $\Box \alpha \stackrel{\text{def}}{=} \neg \Diamond \neg \alpha$, and the "next" operator by $\bigcirc \alpha \stackrel{\text{def}}{=} 1; \alpha$.

A reaction describes an activation requirement for a system task by an expression of the form:

$$[\text{activating-event} \Rightarrow \text{response-task}] : \text{aborting-event} \leq \text{deadline}.$$

The activating and aborting events are normal event expressions, the response-task is the name of a system task declared in the act, and the deadline is a time expression. The activating event must be explicitly specified in a reaction, and is considered a cause of the response task in the real-time model for MASS events. In contrast, the aborting event and the deadline are both optional.

The intended meaning of a reaction is that the response task has to be activated following each occurrence of the activating event, and the termination of its execution must be observed within the duration designated by the deadline. However, MASS tasks are not executed instantaneously and therefore the termination of the response task cannot be observed but strictly after the occurrence of the activating event. A task terminates normally if its execution is completed no later than the first occurrence of the aborting event. Otherwise, the task execution is aborted at the occurrence of the aborting event, returning the value "!". If an execution exceeds the deadline, the system fails (the actual implementation of a system failure is left as a design decision).

For example, consider the reaction $[\text{TrainOut} \Rightarrow \text{GateOpen}] : \text{TrainIn} \leq 10\text{sec}$ where *TrainIn* and *TrainOut* are environment tasks that indicate, respectively, the entrance and exit of a train in a railroad crossing, and the task *GateOpen* denotes the function that moves the gate up. The meaning of this reaction is that upon each occurrence of the event *TrainOut*, the task *GateOpen* should be activated, and its execution must be completed within 10 seconds. However, the execution of *GateOpen* is aborted, generating the event $\text{GateOpen} = !$, if the event *TrainIn* occurs while the gate is opening.

TimeBase. The TimeBase declaration specifies a time unit which is used as a concrete measure for the interpretation of the event "1". Operationally, this quantity defines the sampling rate of the synchronous execution environment (see below). Thus, it determines the resolution at which events can be observed, and therefore it affects the extent to which activation requirements can be satisfied.

Execution environment. The formal semantics of an act is expressed by a regular expression over runs. In practice, the regular expression is transformed into a finite automaton that monitors the occurrences of events, and reacts synchronously by activation and abortion of system tasks. The entire execution environment consists of a reactive executive and a functions executive that run concurrently the act-automaton and the execution of the activated functions, respectively.

The reactive executive runs synchronously at the time-base rate. At each time instant t_i , the input to the automaton is an observation set that consists of those tasks whose executions terminated at the period $[t_{i-1}, t_i)$. Terminations of environment tasks are reported by the plant sensors. For system tasks, the termination is reported by the functions executive. The automaton evaluates the observation set in order to identify occurrences of activating and aborting events specified by the act's reactions, and respectively generates activation and abortion commands for the functions executive.

Virtual events. MASS contains an additional task type called *virtual* (also declared within the Tasks section). The basic events of a virtual task are identified with occurrences of events generated by other, previously defined, tasks. A virtual task has no executions of its own; its behavior merely reflects the executions of the tasks used to define its basic events. Thus, a virtual task can be used only in specifications of activating and aborting events. The basic form of a virtual task declaration is v at α where v is the virtual task name and α is a MASS event (called the marking event).

A virtual event is defined to occur at the time instants that end the intervals designating the occurrences of its marking event. A marking event may itself be defined in terms of virtual events (cyclic definitions are eliminated at compile time).

Virtual tasks are a useful means of abstraction, as they reduce a complex event expression to a basic event. In the general case, recursion enables the representation of regular expressions. For instance, a periodic event that occurs every 100ms can be defined as follows.

Every100ms at startup V (Every100ms;100ms)

2.1 Modularity: Refinement, Composition, and Plays

Usually, the design of a system evolves through a number of abstraction levels, each of which adds design decisions that concretize the implementation towards a machine-executable program. MASS provides two mechanisms that enable a hierarchical modular representation of the controller design by a structure of acts.

- Task *refinement* associates a task with an act that is considered its implementation (concrete examples are given in the specification of the cruise controller in the next section). Operationally, each activation of the refined task causes a separate execution of the refinement.
- A *Composition* is a representation of a task q by a of acts $A_1, \dots, A_k$, expressed as:

Act q is $(A_1 || \dots || A_k)$ End

Operationally, the acts A_i become active simultaneously with each activation of q . A system task declared in one act may be declared as an environment task in any of the other acts, enabling these acts to react to the events generated by that task.

Task refinement and composition enable a hierarchical modular construction of a system. The process starts with an act specifying the operation of the system at a top-level view. Then, system tasks are

separately refined by (a composition of) acts that elaborate their operations in terms of lower level tasks. The process can be iteratively applied to tasks in lower levels until all system tasks are represented by functional computations. A set of acts that establish a hierarchical structure of refinements is called a *play*.

3 Specifying Large Systems with MASS

In this section, we demonstrate the applicability of MASS to the specification of large real-time systems. We present the specification of a cruise-control system that evolves through iterative hierarchical system refinement. This example is extensively worked out in the literature to demonstrate real-time specification frameworks (see Shaw [8] for survey). Thus, it is possible to compare MASS with other approaches.

3.1 Automatic Cruise Control

The Automatic Cruise Control (ACC) is intended to control the speed of a car according to the driver's instructions. The interface with the driver consists of a master switch (on/off), a 3-state speed-command lever (decrease, maintain, increase), a resume button, and the gas and brake pedals. The ACC takes over the speed control whenever the master switch is turned on, provided the car engine is working. The control is released either if the engine goes off, or the master switch is turned off.

While active, the ACC operates to maintain the car speed. The driver may instruct the system to increase or decrease the maintained speed by holding the speed-command lever at the corresponding position until the required speed is attained. The control operation is immediately suspended in case either the brake or the gas pedal are pressed. In this case, the driver may return control to the ACC by pressing the resume button (in which case the ACC returns to maintain the suspended speed, provided the brake and gas pedals are not pressed).

3.2 Hierarchical System Refinement

The specification of the ACC system is developed through iterative hierarchical system refinement. This method suggests to start a large system specification by refining the main thread of the system behavior. Everything else, though known to be part of the system, is considered part of the environment. In succeeding iterations, the structure is broadened by transferring environment activities into the system, and refining their behaviors. At any stage of the development the specification is amenable to simulation where the unspecified behavior is considered to be part of the environment.

In a development step, a task is refined either into a composition of tasks that represents a partition into concurrent activities, or by an act that specifies its design in terms of lower-level activities, also represented by tasks.

In the case of refinement by composition, we can independently proceed with any one of the constituent tasks. However, the partition of a composition need not specify all the constituents in order to be able to proceed with a refinement of a certain task in the composition. This is possible, since in an act specification we make no assumption regarding the source of an environment task; whether it is external to the whole system, or a system task in another act of the play. Thus one may declare environment tasks, known to be part of another subsystem, even though the corresponding act has not been specified yet. Similarly, in case of task refinement by an act, one may proceed with the refinement of any of the system tasks that constitute the act independently of the refinement of other system tasks.

A refinement iteration may be taken to any desired extent, yielding a complete specification in the sense that no assumptions are made regarding unrefined tasks. In further iterations we go over each specification level, and either expand unrefined system tasks, or extend a composition with new acts that specify the behavior of tasks that were, until now, considered part of the environment. We proceed with that process until all system tasks are decomposed into basic components.

In the following subsections, this method is illustrated by working out the specification of the ACC.

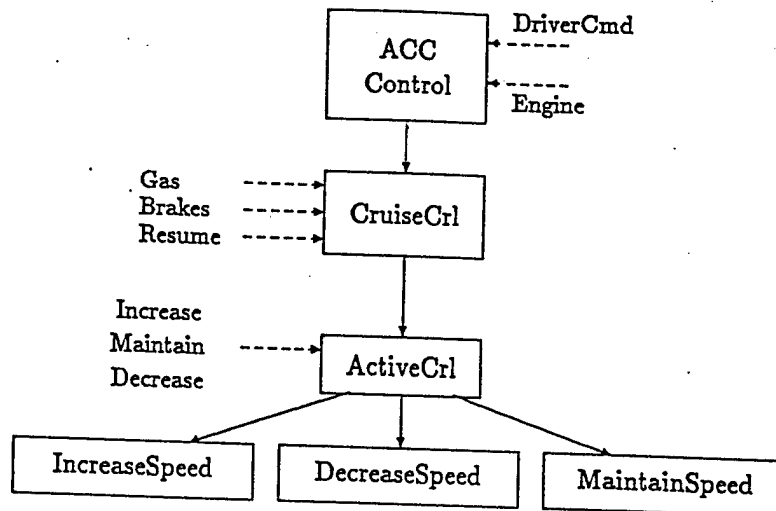


Figure 2: Main control-thread of ACC

3.3 Main Tread Specification

We start with the specification of what seems to us to be the main thread of the system operation (Fig. 2). At the top level, the act ACC-Control controls the activation of the cruise-control function, denoted by the task CruiseCtrl, according to the driver's instructions and the state of the engine.

Act ACC-Control is

Tasks

environment Engine:→{off, on}, DriverCmd:→{start, stop}

system CruiseCtrl

virtual EngineOn at (Engine=on;¬◇Engine=off)

Reactions

[DriverCmd=start ∧ EngineOn ⇒ CruiseCtrl] : Engine=off∨DriverCmd=stop

End

In this act, the driver commands and the engine status are represented by the environment tasks DriverCmd and Engine, respectively. The reaction specifies CruiseCtrl to be activated whenever the driver instructs the system to take over, provided the engine is turned on (denoted by the virtual task EngineOn). The task is aborted either due to a corresponding driver command, or when the engine is turned off (CruiseCtrl is designed as a non-self-terminating operation, thus it must be externally aborted in order to be stopped).

The task EngineOn demonstrates a typical usage of virtual events, as phase designators. This declaration sustains the fact that the engine is turned on by generating the event EngineOn repeatedly at every time instant as long as the engine remains in this phase.

Note that at this stage we are not concerned with how the driver commands and the engine status are monitored, therefore they are represented by environment tasks. At a later stage, we can independently (in different acts) specify the activation requirements for these operations (as indeed we do), and compose them with the control operation. Also note that the time domain is not specified. It is expected to be added at a later stage after a timing analysis of the activation requirements.

The task CruiseCtrl is further refined as follows.

Act CruiseCtrl

Tasks

environment Brakes, Gas, Resume

```

system ActiveCrl
virtual
  Suspend at (CruiseCmd;first(Brakes V Gas)),
  CruiseCmd at (startup V ((Suspend;first(Resume^¬Brakes)))
Reactions [ CruiseCmd ⇒ ActiveCrl ] :Suspend
End

```

first(α) is defined by $(*; \alpha) \wedge \neg \Diamond(\alpha; 1)$, indicating an interval that ends with the first occurrence of α .

The virtual task Suspend designates the transition from active to suspended control (due to gas or brake pedal press, represented by the environment tasks Brakes and Gas). The virtual task CruiseCmd designates the transition to active control, either at startup or due to a press of the resume button (represented by the environment task Resume) in a suspended state while the brakes are not pressed. The task ActiveCrl denotes the ACC active control operation.

Next we refine ActiveCrl as follows.

```

Act ActiveCrl
Tasks
  environment Decrease, Maintain, Increase
  system MaintainSpeed, DecreaseSpeed, IncreaseSpeed
Reactions
  [ (startup V Maintain) ⇒ MaintainSpeed ] :Decrease V Increase
  [ Decrease ⇒ DecreaseSpeed ] :Maintain
  [ Increase ⇒ IncreaseSpeed ] :Maintain
End

```

The environment tasks Increase, Decrease, and Maintain designate the corresponding changes in the lever position, and the task CurrentSpeed provides the car speed at every time instant (it is the same task as declared in CruiseCrl). The system tasks MaintainSpeed, DecreaseSpeed, IncreaseSpeed implement the control operations corresponding to the current lever position.

The concrete behavior of the control operations is given by the refinements below. The act MaintainSpeed operates a control-loop, at 10Hz rate,² to maintain the car speed as recorded at the act activation (the first reaction).

```

Act MaintainSpeed
Tasks
  system SpeedControl, ReadSpeed
Reactions
  [ startup ⇒ ReadSpeed ] ≤70ms
  [ Every100ms ⇒ SpeedControl ] ≤20ms
End

```

The acts, DecreaseSpeed and IncreaseSpeed operate to increase and decrease the car speed by moving the throttle up and down, respectively, in an open loop.

```

Act DecreaseSpeed
Tasks
  system RetractThrottle
Reactions
  [ Every100ms ⇒ RetractThrottle ] ≤20ms
End

```

```

Act IncreaseSpeed
Tasks
  system AdvanceThrottle
Reactions
  [ Every100ms ⇒ AdvanceThrottle ] ≤20ms
End

```

²Events designating periodic signals are not explicitly declared in acts. They are assumed to be pre-defined by virtual tasks. For instance, a 1Hz signal is defined by Every1sec at startup V (Every1sec;1sec).

The last acts deal with basic activation requirements (namely, the system tasks can not be further refined). Hence, the main thread of the system refinement has been completed. In the next section, we describe the next iteration where the play is completed with horizontal (composition) acts.

3.4 Completion of the Play

At this stage, after the main thread of the system operation had been specified, we elaborate declarations of environment tasks, and combine them into the entire system specification. At the top level of the specification, we define the acts EngineMon, DriverCmdMon, and SpeedMon.

The act DriverCmdMon encapsulates the master-switch that turns the ACC on and off. This act is responsible for generating the events denoted by DriverCmd (used by the act ACC-Control).

```

Act DriverCmdMon
  Tasks
    environment MasterSwOff, MasterSwOn
    virtual DriverCmd:→{ start, stop }
      where DriverCmd=start at (0.2sec;MasterSwOn)
            DriverCmd=stop at MasterSwOff
  End

```

The environment tasks MasterSwOff and MasterSwOn represent hardware interrupts generated upon turning the master switch off and on, respectively. DriverCmd is declared as a virtual task considered an abstraction of the interrupts. Note that every activation commands that occur within the first 0.2 seconds of the system operation are ignored. This delay, although not specified in the ACC requirements, is necessary in order to allow speed computation and engine monitoring prior to entering the active-control state.

This act is superfluous since we could directly use MasterSwOff and MasterSwOn in ACC-Control. However, in the context of a whole system development and maintenance, it is good design practice to hide the specific implementation of the sensor. For instance, in a later stage of the development, we could decide to modify the implementation by using polling instead of interrupts (see below). With the suggested design, this would not affect the acts that use the task DriverCmd.

The act EngineMon monitors the engine in order to detect the events indicating the engine being turned on and off (represented by the task Engine which is used by the act ACC-Control).

```

Act EngineMon
  Tasks
    system EngineState:→{ off, on }
    virtual Engine:→{ off, on }
      where Engine=off at Startup V (Engine=on;first(EngineState=off))
            Engine=on at (Engine=off;first(EngineState=on))
  Reactions [ Every20ms ⇒ EngineState ] < 10ms
  End

```

Here, we prefer an implementation by polling (mentioned above). The task EngineState periodically samples the state of the engine, and Engine is declared as a virtual task that designates the changes in the state of the engine operation.

In order to compose the additional acts with ACC-Control we add an upper level act, ACC, declared as follows.

```

Act ACC is ( ACC-Control || DriverCmdMon || EngineMon ) End.

```

At the next level, we define the acts BrakesMon, GasMon and ResumeMon that encapsulate, respectively, the events Brakes, Gas and Resume. We assume that these events are generated by interrupts, hence the acts are very simple (similar to DriverCmdMon).

However, the composition of these acts into the play turns out to be somewhat tedious. Normally, we would need to rename the act CruiseCrl, say by MainCruiseCrl, and then introduce the definition

Act CruiseCrl is (MainCruiseCrl || BrakesMon || GasMon || ResumeMon) End.

This procedure is undesirable for a number of reasons.

- It enforces modifications of already existing part of the specification.
- It inflates the play with superfluous hierarchical levels (not contributing levels of essential information).
- It presents at the same abstraction level acts that are not equally significant in the specification of the system behavior at this level. Such a presentation seems unnatural, and blurs the picture.

Therefore, we allow naming a composition by one of the constituent act. This form emphasizes the subsystem which seems to be central (in the developer's view), and does not force a designer to have a complete view of the system structure at the initial stage.

Thus, in our case, we define the composition

Act CruiseCrl is (CruiseCrl || BrakesMon || GasMon || ResumeMon) End.

Finally, we define the act LeverMon that generates the events Decrease, Maintain, and Increase, which indicate the corresponding transitions in the lever positions. The implementation of this act relies on periodic sampling of the lever and identification of the state transitions by virtual tasks.

Act LeverMon

Tasks

system LeverPosition:→{ down, mid, up }

virtual

Maintain at startupV ((Increase V Decrease);first(LeverPosition=mid)),

Increase at (Maintain;first(LeverPosition=up)),

Decrease at (Maintain;first(LeverPosition=down))

Reactions [Every200ms ⇒ LeverPosition] < 10ms

End

Here, again, we employ the relaxed form of composition, and define

Act ActiveCrl is (ActiveCrl || LeverMon) End.

4 Related Work and Discussion

MASS is a synchronous language that monitors the execution of asynchronous computations under real-time constraints. This execution paradigm, which overcomes the essential limitation of infinitesimally short computations assumed by the synchronous model [1], is enabled by the idea of identifying system events with the terminations of the computations.

CRP introduced by Berry et al. [3] presents an alternative approach. The language extends ESTEREL [2] with a special construct, *exec L:P*, which implicitly associates the label *L* with the events (signals) denoting the start, termination, and abortion of an execution of *P*. This approach is inherently restricted since once engaged with an execution, a program cannot respond to additional activations. For instance, if *P* is an asynchronous computation, a requirement like "activate *P* upon every occurrence of the event α " is not expressible in this formalism.

Synchronous Eifel (previously called *Embedded Eifel* [4]) employs an execution environment similar to MASS, but without monitoring asynchronous computations. The reactive behavior is specified in the synchronous language ESTEREL, augmented with a special construct *schedule(f)* where *f* is a function. Functions scheduled this way, called background services, are run asynchronously (in a non-preemptive manner) in the time slots between the termination of the synchronous computations and the next time instant, with no

deadlines. A background service can communicate with the reactive part by a special command that adds a signal to the next time instant.

MASS is an executable language in the full operational sense. A similar approach is also taken in the design of SAFE [6], which is a procedural real-time programming language with interval temporal-logic semantics (however, SAFE does not support concurrency).

5 Conclusion

The main contribution of the PLOT/MASS framework is the idea of associating each primitive event in a real-time system with a concrete function, and interpreting every termination of the function execution as an occurrence of the event. This idea gives rise to a real-time specification framework that enables the design of real-time systems, and formal reasoning about the behavior of asynchronous functions whose executions are controlled synchronously.

MASS, introduces a new declarative activation-oriented specification approach, which we believe is natural for real-time system representation, and easy for system designers to understand and use. Another novelty of MASS is the concept of activation refinement, which provides an essential means for modular development and reasoning.

Finally, the fact that events and computations are semantically related provides for a complete separation of the specification of the reactive and algorithmic aspects of a system, an important software-engineering concern.

References

- [1] A. Benveniste and P. Le Guernic. A denotational theory of synchronous reactive systems. *Information and Computation*, 99(2):192-230, August 1992.
- [2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, (19):87-152, 1992.
- [3] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *20th Annual ACM Symp. on Principles of Programming Languages*, Charleston, Virginia, 1993.
- [4] R. Budde. ee - the design and programming language embedded eifel. Technical Report version 0.9, GMD, 1997.
- [5] V. Gafni. *A Specification Framework for Real-time Systems*. PhD thesis, Tel-Aviv University, 1997.
- [6] R. W. S. Hale. The real-time programming language, safe: Interval semantics and derived laws. Technical Report CRC-039, Stanford Research Inst., 1993.
- [7] B. Moszkowski. *Reasoning about Digital Circuits*. PhD thesis, Stanford University, 1983.
- [8] M. Shaw. Making choices: A comparison of styles for software architecture. *IEEE Software*, 12(6):27-41, November 1995.
- [9] G. Zehavi. Specification and implementation of robotic applications using mass, 1993.

Parametric Approach to the Specification and Analysis of Real-time System Designs based on ACSR-VP *

Hee-Hwan Kwak, Insup Lee, and Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA

lee@cis.upenn.edu, {heekwak,sokolsky}@saul.cis.upenn.edu

March 19, 1999

Abstract

To engineer reliable real-time systems, it is desirable to discover timing anomalies early in the development process. However, there is little work addressing the problem of accurately predicting timing properties of real-time systems before implementations are developed. This paper describes an approach to the specification and analysis of scheduling problems of real-time systems. The method is based on ACSR-VP, which is an extension of ACSR, a real-time process algebra, with value-passing capabilities. Combined with the existing features of ACSR for representing time, synchronization and resource requirements, ACSR-VP can be used to describe an instance of a scheduling problem as a process that has parameters of the problem as free variables. The specification is analyzed by means of a symbolic algorithm. The outcome of the analysis is a set of equations and a solution to which yields the values of the parameters that make the system schedulable. These equations can be solved using integer programming or constraint logic programming. The paper presents the theory of ACSR-VP briefly and an example of the period assignment problem for rate-monotonic scheduling. We also explain our current tool implementation effort and plan for incorporating it into the existing toolset, PARAGON.

1 Introduction

The desire to automate or incorporate intelligent controllers into control systems has lead to rapid growth in the demand for real-time software systems. Moreover, these systems are becoming increasingly complex and require careful design analysis to ensure reliability before implementation. Recently, there has been much work on formal methods for the specification and analysis of real-time systems [8, 12]. Most of the work assumes that various real-time systems attributes, such as execution time, release time, priorities, etc., are fixed *a priori* and the goal is to determine whether a system with all these known attributes would meet required safety properties. One example of safety property is schedulability analysis; that is, to determine whether or not a given set of real-time tasks under a particular scheduling discipline can meet all of its timing constraints.

The pioneering work by Liu and Layland [17] derives schedulability conditions for rate-monotonic scheduling and earliest-deadline-first scheduling. Since then, much work on schedulability analysis has been done which includes various extensions of these results [11, 28, 25, 4, 26, 22, 18, 3]. Each of these extensions expands the applicability of schedulability analysis to real-time task models with different assumptions. In particular, there has been much advance in scheduling theory to address uncertain nature of timing attributes at the design phase of a real-time system. This problem is complicated because it is not sufficient to consider the worst case timing values for schedulability analysis. For example, scheduling anomalies can occur even when there is only one processor and jobs have variable execution times and are nonpreemptable. Also for preemptable jobs with one processor, scheduling anomalies can occur when jobs have arbitrary release times and share resources. These scheduling anomalies make the problem of validating a priority-driven system difficult. Clearly, exhaustive simulation or testing is not practical in general except for small systems of practical interest. There have been many different heuristics developed to solve some of these general schedulability analysis problems. However, each algorithm is problem specific and thus when a problem is modified, one has to develop new heuristics.

*This research was supported in part by ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, AFOSR F49620-96-1-0204, NSF CCR-9619910, and ONR N00014-97-1-0505.

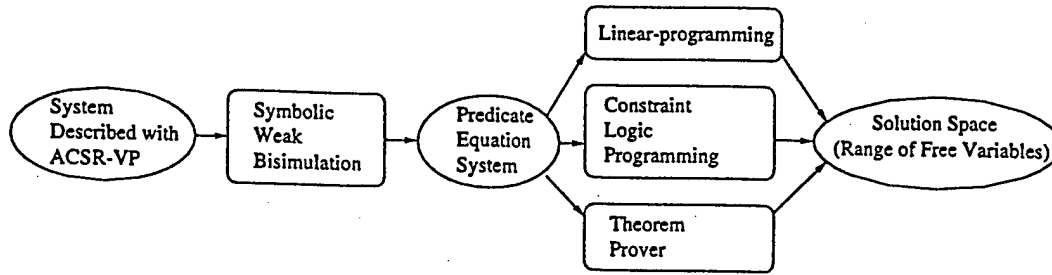


Figure 1: Overview of the Framework

In this paper, we describe a framework that allows one to model scheduling analysis problems with variable release and execution times, relative timing constraints, precedence relations, dynamic priorities, multiprocessors etc. Our approach is based on ACSR-VP and symbolic bisimulation algorithm.

ACSR (Algebra of Communicating Shared Resources) [14], is a discrete real-time process algebra. ACSR has several notions, such as resources, static priorities, exceptions, and interrupts, which are essential in modeling real-time systems. ACSR-VP is an extension of ACSR with value-passing and parameterized processes to be able to model real-time systems with variable timing attributes and dynamic priorities. In addition, symbolic bisimulation for ACSR-VP has been defined. ACSR-VP without symbolic bisimulation has been applied to the simple schedulability analysis problem [5], by assuming that all parameters are ground, i.e., constants. However, it is not possible to use the technique described in [5] to solve the general schedulability analysis problem with unknown timing parameters.

Figure 1 shows the overall structure of our approach. We specify a real-time system with unknown timing or priority parameters in ACSR-VP. For the schedulability analysis of the specified system, we check symbolically whether or not it is bisimilar to a process idling forever. The result is a set of predicate equations, which can be solved using widely available linear-programming or constraint-programming techniques. The solution to the set of equations identifies, if exists, under what values of unknown parameters the system becomes schedulable. To support the effective use of the the symbolic ACSR-VP analysis, we are developing a tool and planning to integrate into PARAGON [27], a toolset with graphical interface to support the use of ACSR.

The rest of the paper is organized as follows. Sections 2 overviews the theory of the underlying formal method, ACSR-VP, and introduce symbolic bisimulation for ACSR-VP expressions. Section 3 gives a specification of a scheduling problem, namely the *period assignment problem* and illustrates how to analyze an instances of this problem. Section 4 briefly describes the PARAGON toolset and its support for value-passing specifications, and outlines the incorporation of ACSR-VP into the toolset. We conclude with a summary and an outline of future work in Section 5.

2 ACSR-VP

ACSR-VP extends the process algebra ACSR [14] by allowing values to be communicated along communication channels. In this section we present ACSR-VP concentrating on its value-passing capabilities. We refer to the above papers for additional information on ACSR.

We assume a set of variables X ranged over by x, y , a set of values V ranged over by v , and a set of labels L ranged over by c, d . Moreover, we assume a set $Expr$ of expressions (which includes arithmetic expressions) and we let $BExpr \subset Expr$ be the subset containing boolean expressions. We let e and b range over $Expr$ and $BExpr$ respectively, and we write $\vec{z}$ for a tuple $z_1, \dots, z_n$ of syntactic entities.

ACSR-VP has two types of actions: instantaneous communication and timed resource access. Access to resources and communication channels is governed by priorities. A priority expression p is attached to every communication event and resource access. A partial order on the set of events and actions, the preemption relation, allows one to model preemption of lower-priority activities by higher-priority ones.

Instantaneous actions, called *events*, provide the basic synchronization and communication primitives in the process algebra. An event is denoted as a pair (i, e_p) representing execution of action i at priority e_p , where i ranges over τ , the idle action, $c?x$, the input action, and $c!e$, the output action. We use $\mathcal{D}_E$ to denote the domain of events and let λ range over events. We use $l(\lambda)$ and $\pi(\lambda)$ to represent the label and priority, respectively, of the event λ ; e.g., $l((c!x, p)) = c!$ and $\pi((c?x, p)) = c?$. To model resource access, we assume that a system contains a finite set of serially-reusable resources drawn from some set R . An action that consumes one tick of time is drawn from the domain $P(R \times Expr)$ with the restriction that each resource is represented at most once. For example the singleton action $\{(r, e_p)\}$ denotes the use of some resource $r \in R$ at priority

level e_p . The action $\emptyset$ represents idling for one unit of time, since no resource is consumed. We let $\mathcal{D}_R$ to denote the domain of timed actions with A, B , to range over $\mathcal{D}_R$. We define $\rho(A)$ to be the set of the resources used by action A , for example $\rho(\{(r_1, p_1), (r_2, p_2)\}) = \{r_1, r_2\}$. We also use $\pi_r(A)$ to denote the priority level of the use of the resource r in the action A ; e.g., $\pi_{r_1}(\{(r_1, p_1), (r_2, p_2)\}) = p_1$, and write $\pi_r(A) = 0$ if $r \notin \rho(A)$. The entire domain of actions is denoted by $\mathcal{D} = \mathcal{D}_R \cup \mathcal{D}_E$, and we let α, β range over $\mathcal{D}$. We let P, Q range over ACSR-VP processes and we assume a set of process constants ranged over by C . The following grammar describes the syntax of ACSR-VP processes:

$$P ::= \text{NIL} \mid A : P \mid \lambda.P \mid P + P \mid P \parallel P \mid \\ b \rightarrow P \mid P \setminus F \mid [P]_I \mid C(\vec{x}).$$

In the input-prefixed process $(c?x, e).P$ the occurrences of variable x is bound. We write $\text{fv}(P)$ for the set of free variables of P . Each agent constant C has an associated definition $C(\vec{x}) \stackrel{\text{def}}{=} P$ where $\text{fv}(P) \subseteq \vec{x}$ and $\vec{x}$ are pairwise distinct. We note that in an input prefix $(c?x, e).P$, e should not contain the bound variable x , although x may occur in P .

An informal explanation of ACSR-VP constructs follows: The process NIL represents the inactive process. There are two prefix operators, corresponding to the two types of actions. The first, $A : P$, executes a resource-consuming action during the first time unit and proceeds to process P . On the other hand $\lambda.P$, executes the instantaneous event λ and proceeds to P . The process $P + Q$ represents a nondeterministic choice between the two summands. The process $P \parallel Q$ describes the concurrent composition of P and Q : the component processes may proceed independently or interact with one another while executing instantaneous events, and they synchronize on timed actions. Process $b \rightarrow P$ represents the conditional process: it performs as P if boolean expression b evaluates to *true* and as NIL otherwise. In $P \setminus F$, where $F \subseteq L$, the scope of labels in F is restricted to process P : components of P may use these labels to interact with one another but not with P 's environment. The construct $[P]_I$, $I \subseteq R$, produces a process that reserves the use of resources in I for itself, extending every action A in P with resources in $I - \rho(A)$ at priority 0.

The semantics of ACSR-VP processes may be provided as a labeled transition system, similarly to that of ACSR. It additionally makes use of the following ideas: Process $(c!e_1, e_2).P$ transmits the value obtained by evaluating expression e_1 along channel c , with priority the value of expression e_2 , and then behaves like P . Process $(c?v, p).P$ receives a value v from communication channel c and then behaves like $P[v/x]$, that is P with v substituted for variable x . In the concurrent composition $(c?x, p_1).P_1 \parallel (c!v, p_2).P_2$, the two components of the parallel composition may synchronize with each other on channel c resulting in the transmission of value v and producing an event $(\tau, p_1 + p_2)$.

2.1 Unprioritized Symbolic Graphs with Assignment

Consider the simple ACSR-VP process $P \stackrel{\text{def}}{=} (in?x, 1).(out!x, 1).\text{NIL}$ that receives a value along channel in and then outputs it on channel out , and where x ranges over integers. According to traditional methods for providing semantic models for concurrent processes, using transition graphs, process P in infinite branching, as it can engage in the transition $(in?n, 1)$ for every integer n . As a result standard techniques for analysis and verification cannot be applied to such processes.

Several approaches have been proposed to deal with this problem for various subclasses of value-passing processes [9, 16, 20, 13]. One of these advocates the use of *symbolic* semantics for providing finite representations of value-passing processes. This is achieved by taking a more conceptual view of value-passing than the one employed above. More specifically consider again process P . A description of its behavior can be sufficiently captured by exactly two actions: an input of an integer followed by the output of this integer. Based on this idea the notion of symbolic transition graphs [9] and transition graphs with assignment [16] were proposed and shown to capture a considerable class of processes.

In this section we present symbolic graphs with assignment for ACSR-VP processes. As it is not the intention of the paper to present in detail the process-calculus theory of this work, we only give an overview of the model and we refer to [13] for a complete discussion.

2.2 Symbolic Graph with Assignment

The notion of a *substitution*, which we also call *assignment*, is defined as follows. A *substitution* is any function $\theta: X \rightarrow Expr$, such that $\theta(x) \neq x$ for a finite number of $x \in X$. Given a substitution θ , the *support* (or *domain*) of θ is the set of variables $D(\theta) = \{x \mid \theta(x) \neq x\}$. A substitution whose support is empty is called the *identity substitution*, and is denoted by Id . When $|D(\theta)| = 1$, we use $[\theta(x)/x]$ for the substitution θ . Given two substitutions θ and σ , the *composition* of θ and σ is the substitution denoted by $\theta; \sigma$ such that for every variable x , $\theta; \sigma(x) = \sigma(\theta(x))$. We often write $\theta\sigma$ for $\theta; \sigma$.

An SGA is a rooted directed graph where each node n has an associated finite set of free variables $\text{fv}(n)$ and each edge is labeled by a guarded action with assignment [16, 23]. Note that a node in SGA is a ACSR-VP term.

Definition 2.1 (SGA) A Symbolic Graph with Assignment (SGA) for ACSR-VP is a rooted directed graph where each node n has an associated ACSR-VP term and each edge is labeled by boolean, action, assignment, (b, α, θ) . $\square$

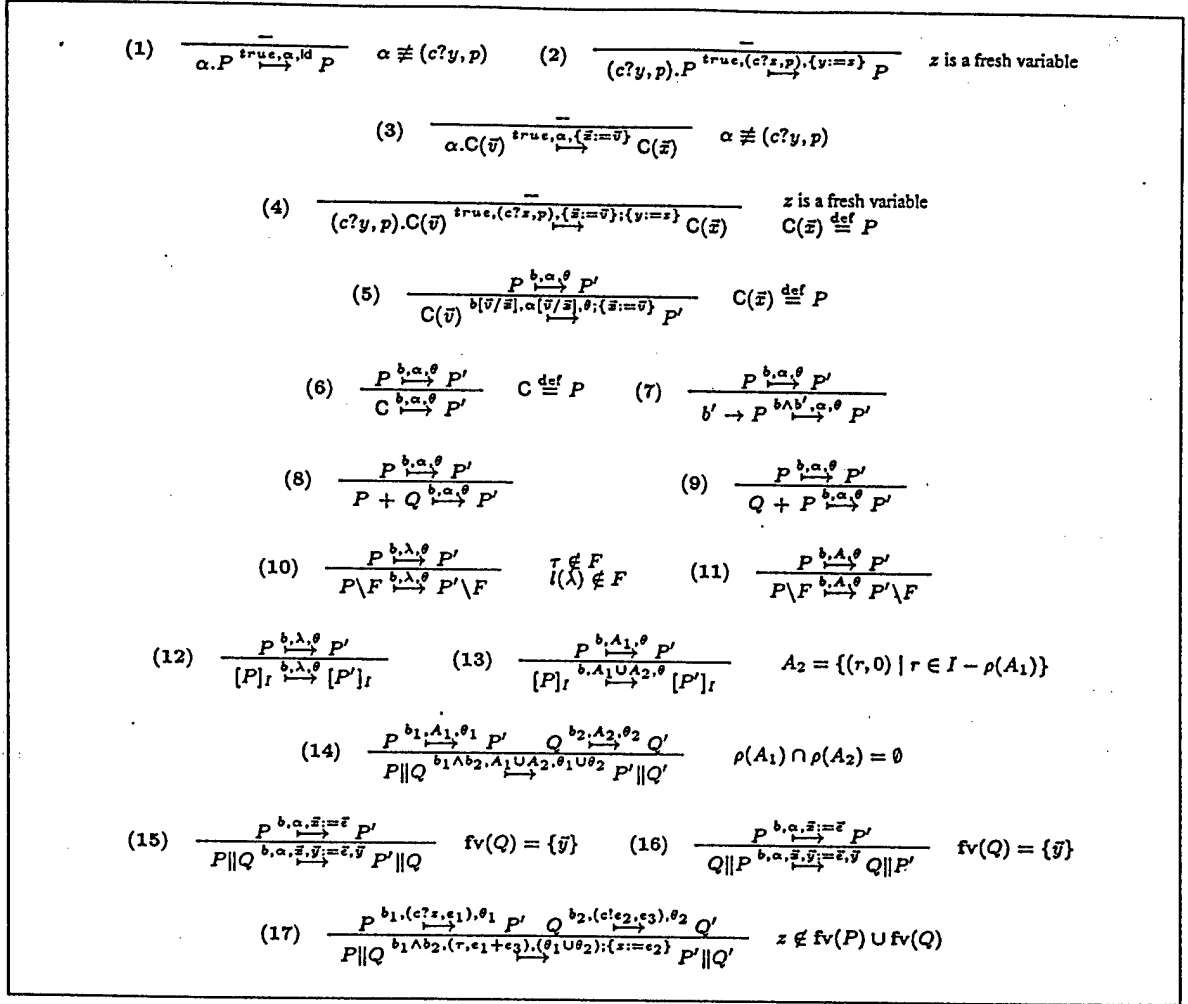
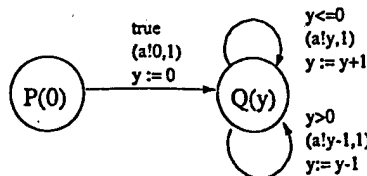


Figure 2: Rules for constructing Symbolic Graphs with Assignment

Given an ACSR-VP term, a SGA can be generated using the rules in Figure 2. Transition $P \xrightarrow{b, \alpha, \theta} P'$ denotes that given the truth of boolean expression b , P can evolve to P' by performing actions α and putting into effect the assignment θ . The interpretation of these rules is straightforward and we explain them by an example: Consider the following process. Process $P(0)$ can output the sequence of events $a!0$ infinitely many times.

$$\begin{aligned} P(x) &\stackrel{\text{def}}{=} (a!x, 1).Q(x) \\ Q(y) &\stackrel{\text{def}}{=} (y \leq 0) \rightarrow (a!y, 1).Q(y+1) \\ &\quad + (y > 0) \rightarrow (a!y-1, 1).Q(y-1) \end{aligned}$$

Following SGA represents the process $P(0)$.



One possible interpretation of our SGA can be given along the lines of programming languages: Process P can be thought of as a procedure, so that $P(0)$ represents a call to P with actual parameter 0 which is accepted by P with formal parameter x declared in P 's body. According to its definition, P outputs $a!0$ and calls process Q with actual parameter 0. Process Q then checks the validity of condition $y \leq 0$ or $y > 0$. If $y \leq 0$ is satisfied, process Q outputs $a!0$ and calls Q with actual parameter $y + 1$, where the value of y is 0 in this case. Similar reasoning can be applied for the condition $y > 0$. We believe that this interpretation, being similar to that of function calls and parameter passing in programming languages, is an intuitive way of interpreting the ACSR-VP terms.

2.3 The prioritized Symbolic Transition System

We have illustrated how ACSR-VP processes can be given finite representations as SGA's via the symbolic transition relation $\mapsto$. However, this relation makes no arbitration between actions with respect to their priorities. To achieve this, we refine the relation $\mapsto$ to obtain the prioritized symbolic transition system $\mapsto_\pi$. This is based on the notion of *preemption* which incorporates our treatment of priority, and in particular on relation $\succ$, the *preemptive relation*, a transitive, irreflexive relation on actions [2]. Then for two actions α and β , $\alpha \succ \beta$ denotes that α preempts β , which implies that in any real-time system, if there is a choice between the two actions, α will always be executed. For example $(c?x, 2) \succ (c?x, 1)$ and $\{(r, 2)\} \succ \{(r, 0)\}$.

Extending the notion of preemption in the value-passing setting involves dealing with the presence of free variables in process descriptions. For example, given actions $\alpha = (c?x, y_1)$ and $\beta = (c?x, y_2)$, whether $\alpha \succ \beta$ or $\beta \succ \alpha$ depends on the values to which variables y_1 and y_2 are instantiated. This idea can easily be incorporated to yield the prioritized transition relation $\mapsto_\pi$. For the precise definition we refer the reader to [13]. We illustrate this with an example. Consider process P :

$$\begin{aligned} P(x) &\stackrel{\text{def}}{=} (a?y, 1).P'(x, y) \\ P'(x, y) &\stackrel{\text{def}}{=} (y \leq 1) \rightarrow (a!(x+y), y).NIL \\ &\quad + (y \leq 2) \rightarrow (a!(x+y), 2).NIL \end{aligned}$$

Figure 3 shows the unprioritized SGA for P and its prioritized version, Q . Note that transition $P' \xrightarrow{y \leq 1, (a!(x+y), y), Id} NIL$ is preempted by $P' \xrightarrow{y \leq 2, (a!(x+y), 2), Id} NIL$ since whenever the former is enabled, the latter is also enabled with a higher priority (that is, whenever $y \leq 1$, we have $y \leq 2$ and $y < 2$).



Figure 3: SGA of P and Q

2.4 Weak Bisimulation

Various methods have been proposed for the verification of concurrent processes. Central among them is observational equivalence that allows to compare an implementation with a specification of a given system. Observational equivalence is based on the idea that two equivalent systems exhibit the same behavior at their interfaces with the environment. This requirement was captured formally through the notion of *bisimulation* [19], a binary relation on the states of systems. Two states are bisimilar, if for each single computational step of the one, then there exists an appropriate matching (multiple) step of the other, leading to bisimilar states.

In this setting, bisimulation for symbolic transition graphs is defined in terms of relations parameterized on boolean expressions, of the form $\simeq^b$, where $p \simeq^b q$ if and only if, for each interpretation satisfying boolean b , p and q are bisimilar in the traditional notion. In [13] the authors have proposed weak version of bisimulations for SGA's, that is observational equivalences that abstract away from internal system behavior (both for late and early semantics). Furthermore, algorithms were presented for computing these equivalences. Given two closed processes whose symbolic transition graphs are finite, the algorithm constructs a predicate equation system that corresponds to the most general condition for the two processes to be weakly bisimilar.

Recall process $P(x)$ from Section 2.3. Furthermore, consider the following process with bound variable x' :

$$\begin{aligned}
R(x') &\stackrel{\text{def}}{=} (a?y', 1).R'(x', y') \\
R'(x', y') &\stackrel{\text{def}}{=} (y' \leq 2) \rightarrow (a!(x' + y' + 1), 2).NIL
\end{aligned}$$

The prioritized SGA for R is similar to Q with the exception that after receiving a value via channel a , R outputs value $x' + y' + 1$. Applying the symbolic bisimulation algorithm for processes P and R , we obtain the following predicate equation system.

$$\begin{aligned}
X_{00}(x, x') &= \forall z X_{11}(z, x, x') \\
X_{11}(z, x, x') &= z \leq 2 \rightarrow z \leq 2 \wedge x + z = x' + z + 1 \\
&\wedge z \leq 2 \rightarrow z \leq 2 \wedge x' + z + 1 = x + z
\end{aligned}$$

This equation system can easily be reduced to the equation $X_{00}(x, x') \equiv x = x' + 1$, which allows us to conclude that $P(x)$ and $R(x')$ are bisimilar if and only if $x = x' + 1$ holds. In general, since we are dealing with a domain of linear expressions, predicate equations obtained from the bisimulation algorithm can be solved using integer programming techniques [24].

3 Real-time Scheduling Problems

In this section, we show how a problem of real-time system scheduling can be specified and analyzed using ACSR-VP. According to [29], real-time scheduling problems can be categorized into the following three groups: priority assignment, execution synchronization, and schedulability analysis problems. The priority assignment problem requires assigning priorities to jobs so that the system schedulability is maximized. The execution synchronization problem is the problem of deciding when and how to release jobs so that the precedence constraints are satisfied and the system schedulability, as well as other performance concerns, are optimized. Schedulability analysis problem is the problem of verifying that a system is schedulable, given a certain priority assignment method and execution synchronization method.

Classic examples of solutions to these problems include the rate-monotonic priority assignment problem on a single processor [17]. It uses static priority assignment, where the priority of each job is assigned in the inverse order of period; a job with the shortest period has the highest priority. Deadline-monotonic priority assignment was proposed by [15], where the system has jobs with arbitrary relative deadlines.

The same groups of problems can be considered in the presence of end-to-end scheduling constraints. Gerber *et al.* [7] proposed the method to guarantee a system's end-to-end requirements of real-time systems. In [30], Tindell *et al.* attempted to compute upper bounds on the end-to-end response time. They also proposed priority assignment in distributed system where jobs have end-to-end deadlines. In [1], Bettati studied the problem of scheduling a set of jobs with arbitrary release times and end-to-end deadlines.

Our Approach. We propose to address real-time scheduling problems by means of analysis based on ACSR-VP. In this approach, a specific instance of a problem is specified as an ACSR-VP expression and symbolically analyzed. Figure 4 shows the overall structure of our approach. Rectangles with thick lines represent tools, and ovals in them represents the functional blocks inside tools. Rectangles with curved corner are text artifacts used as input/output for tools. We specify scheduling problems in the real-time system with unknown timing or priority parameters in the restricted form of ACSR-VP. The restricted form of ACSR-VP is defined to ensure that resulting SG (Symbolic Graph without assignment) derived from SGA is finite.

With a given set of ACSR-VP processes in the restricted form, the SGA is generated to capture the semantics of model. There are two paths that lead us to a solution. With the first path, we can generate the finite SG from the SGA and check bisimilarity with an infinite idle process. The result is a boolean expression with unknown parameters. This kind of boolean expression can be solved using integer programming, e.g., Omega Test [21], to find all solutions of the parameters. With the second path, the generated SGA is checked symbolically whether it is bisimilar to an idling process. Here, the result is a set of predicate equations with unknown parameters. This resulting set of equations can then be translated into a constraint logic program or into a boolean formula.

For a real-time scheduling problem, if a solution to a boolean expression or to the set of predicate equations exists, then it identifies under what values of unknown parameters the system becomes schedulable. Thus, the schedulability analysis is performed symbolically. For instance, in the rate-monotonic scheduling shown below, we want to find the periods of jobs to guarantee that a system can be scheduled. We call this problem the *period assignment* problem. In this problem, we let periods be free variables and describe a system as ACSR-VP process terms. These free variables appear in the resulting boolean expression or predicate equations that are generated from the bisimulation algorithm. Solutions for free variables represent the valid ranges of periods of the jobs, which make the system schedulable.

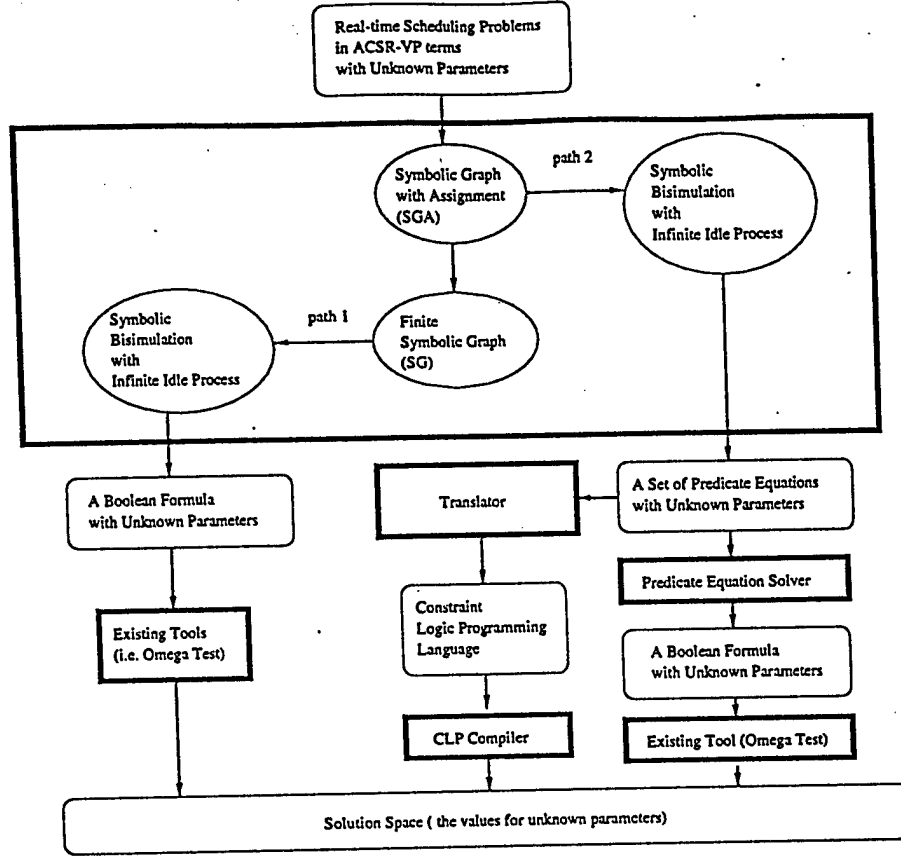


Figure 4: Our Approach to Real-time Scheduling Problem

Our method is expressive to model complex real-time systems in general. Furthermore, it is effective in the sense that the resulting boolean formulas and predicate equations can be solved efficiently. For instance, there has been active research [6] to solve boolean formulas efficiently, and there are existing tools such as omega test [21], which are very fast in practice. For predicate equations, there are constraint programming techniques that are known for solving linear (in)equation constraints efficiently [10, 24]. Furthermore, the size of the SGAs constructed from ACSR-VP terms is significantly smaller than that of Labeled Transition Systems (LTS) constructed from ACSR. Consequently, this greatly reduces the state explosion problem, and thus, we can now model larger systems and solve problems which are not possible using ACSR (and its toolset called PARAGON) due to state explosion.

We now illustrate our approach by showing how to solve a rate-monotonic scheduling problem, known as the *period assignment problem*. Our method of solving this problem is optimal in the sense that if the method can not find a period assignment, then the system cannot be scheduled for any assignment of periods.

Period Assignment Problem for Rate Monotonic Scheduling. We briefly state how rate monotonic scheduling works and show our approach to the period assignment problem. Rate monotonic scheduling is a preemptive static priority driven scheduling algorithm, which works as follows. The priorities of tasks are assigned in the reverse order of lengths of their periods, that is, tasks with shorter periods are assigned higher priorities than tasks with longer periods. Scheduling decisions are made whenever any task becomes ready or whenever a processor becomes idle. At each scheduling decision time, a ready task with the highest priority is executed. The following ACSR-VP process describes a job with unknown period:

$$\begin{aligned}
 Job_i(p_i, s_i, t_i) &\stackrel{\text{def}}{=} (s_i < E_i) \wedge (t_i < D_i) \rightarrow \{(cpu, MAX - p_i)\} : Job_i(p_i, s_i + 1, t_i + 1) \\
 &\quad + (s_i = E_i) \wedge (t_i \leq D_i) \rightarrow Wait(p_i, t_i) \\
 Wait_i(p_i, t_i) &\stackrel{\text{def}}{=} (t_i \leq P_{i,max}) \wedge (t_i < p_i) \rightarrow \emptyset : Wait_i(p_i, t_i + 1) \\
 &\quad + (t_i \leq P_{i,max}) \wedge (t_i = p_i) \rightarrow (\tau, 1).Job_i(p_i, 0, 0)
 \end{aligned}$$

where E_i and D_i represent the constant values for the execution time and deadline of Job_i , respectively. Process $Job_i(p_i, s_i, t_i)$ represents a job with period p_i , which has accumulated s_i units of processing time in the current period. The current period has started t_i time units ago. As long as the job is not finished ($s_i < E_i$) and the current deadline is not over ($t_i < D_i$), the job competes with other jobs for access to the *cpu* resource. The priority of Job_i is $MAX - p_i$, where MAX is the largest possible period. That is, the job with shortest period has the highest priority. If the job is preempted by a higher-priority process, it idles in that time unit. Alternatively, if the job has completed ($s_i = E_i$), it turns into the $Wait_i(p_i, t_i)$ process, which idles until the end of the current period and restarts itself. $P_{i,max}$ represents the possible maximum value for the period of Job_i . In this rate monotonic setting, priorities are unknown since the period of each job is not known.

For a job Job_i where period is known to be P_i , it can be described as follows:

$$\begin{array}{lll}
Job_i(s_i, t_i) & \stackrel{\text{def}}{=} & (s_i < E_i) \wedge (t_i < D_i) \rightarrow \{(cpu, MAX - P_i)\} : Job_i(s_i + 1, t_i + 1) \\
& & + \emptyset : Job_i(s_i, t_i + 1) \\
& + & (s_i = E_i) \wedge (t_i \leq D_i) \rightarrow Wait_i(t_i) \\
Wait_i(t_i) & \stackrel{\text{def}}{=} & (t_i < P_i) \rightarrow \emptyset : Wait_i(t_i + 1) \\
& + & (t_i = P_i) \rightarrow (\tau, 1).Job_i(0, 0)
\end{array}$$

Assuming that, initially, all jobs start at time 0, we can capture the behavior of the whole system as follows.

$$RM(\vec{p}) \stackrel{\text{def}}{=} [Job_1 \parallel \dots \parallel Job_n]_{\{cpu\}}$$

where $Job_i, i \in \{1, \dots, n\}$ can be either $Job_i(p_i, 0, 0)$ if the period is unknown or $Job_i(0, 0)$ otherwise. Symbolic weak bisimulation relation with infinite idle process can be checked by applying the algorithm shown in [13]. The result is a set of predicate equations or a boolean expression if we translate the SGA into the SG. A boolean expression can be solved automatically by the existing integer programming tool.

4 PARAGON Toolset

Our approach to the symbolic analysis of ACSR-VP specifications can be applied effectively to non-trivial problems only if there are good tool supports for specifications in ACSR-VP and analysis algorithms described in the preceding sections. Their usefulness will be enhanced if this tool support is provided within an extensive specification and verification framework, where symbolic analysis can be supplemented by other analysis techniques. Such framework can be provided by extending PARAGON [27], a toolset based on ACSR and other related formalisms.

PARAGON is a toolset for the specification and analysis of distributed resource-bound real-time systems. PARAGON supports both graphical and textual input. Graphical specifications enhance the usability of a formal model, giving a visual representation of hierarchy modules in the system and of interconnections between modules. Graphical specifications in PARAGON are expressed using the GCSR language, based on a real-time process algebra. A GCSR specification is a collection of processes, which consist of nodes, connected by edges. The execution of the system proceeds from node to node along the edges. There are several types of nodes to express sequential behavior of a system module and its resource requirements. In addition to these, a *compound* node provides hierarchy. One or more parallel processes can be placed into a compound node. Interactions between processes in a compound node can be made local to the node, that is, invisible to the processes outside.

For analysis, PARAGON supports several techniques: extensive checking of syntactic consistency constraints; state space exploration, including reachability analysis and deadlock detection; checking equivalence between two specifications; and visual simulation.

PARAGON already supports parameterization in specifications and can deal with value passing. This enables concise specification of arrays of similar components, multiple resources of the same type, and value passing between processes. Event and resource names and process references in an indexed specification can contain multiple indices. Indices may be represented as integers or integer-valued expressions using index variables. The syntax of the current parameterized specifications, although slightly different from that of ACSR-VP, provides for an easy translation between the two formalisms. However, parametric treatment of data values is currently missing in PARAGON. Every parameterized PARAGON specification is equivalent to a non-parameterized one, and handling of parameterization during analysis is done through an "un-parameterizing" translation. This approach is very inefficient, as it creates a separate process for every instantiation of free index variables in the parameterized process, many of which are not necessary for the subsequent analysis. Therefore, it is necessary to use a better internal representation that handles index variables symbolically, such as SGA described in this paper.

5 Conclusions

We have described a formal framework for the specification and analysis of real-time scheduling problems. Our framework is based on ACSR-VP and symbolic bisimulation. The major advantage of our approach is that the same framework can be used for scheduling problems with different assumptions and parameters. In other scheduling-theory based approaches, new analysis algorithms need to be devised for problems with different assumptions since applicability of a particular algorithm is limited to specific system characteristics.

We believe that ACSR-VP is expressive enough to model any real-time system. In particular, our method is appropriate to model many complex real-time systems and can be used to solve the *priority assignment problem*, *execution synchronization problem*, *end-to-end design problem*, and *schedulability analysis problem*. It depends on light-weight formal methods in the sense that resulting predicate equation systems can be solved with existing techniques such as linear programming or constraint programming, which can be solved using linear equation constraints efficiently in practice [24].

The novel aspect of our approach is that parametrized design of a real-time system can be described formally and analyzed automatically, all within a process-algebraic framework. It has often been noted that scheduling work is not adequately integrated with other aspects of real-time system development [3]. Our work is a step toward such an integration, which helps to meet our goal of making the timed process algebra ACSR a useful formalism for supporting the development of reliable real-time systems. Our approach allows the same specification to be subjected to the analysis of both schedulability and functional correctness.

There are several issues that we need to address to make our approach practical. We showed that resulting predicate equation systems can be solved with constraint logic programming or linear programming, but they can be rather complicated. We plan to investigate when resulting equation systems become easy or difficult to solve. In the worst case, we may have to use a more powerful technique such as theorem prover; however, it is not clear whether any reasonable real-time system scheduling problem can result in such a complex equation system. We are currently augmenting PARAGON, the toolset for ACSR, to support the full syntax of ACSR-VP directly and implementing a symbolic bisimulation algorithm. This toolset will allow us to experimentally evaluate the effectiveness of our approach with a number of large scale real-time systems.

References

- [1] R. Bettati. *End-to-end Scheduling to Meet Deadlines in Distributed Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.
- [2] P. Brémont-Grégoire, I. Lee, and R. Gerber. ACSR: An Algebra of Communicating Shared Resources with Dense Time and Priorities. In *Proc. of CONCUR '93*, 1993.
- [3] A. Burns. Preemptive priority-based scheduling: An appropriate engineering approach. In Sang H. Song, editor, *Advances in Real-Time Systems*, chapter 10, pages 225–248. Prentice Hall, 1995.
- [4] M. Chen and K. Lin. Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems. *Real-Time Systems*, 2(4):325–346, 1990.
- [5] J-Y. Choi, I. Lee, and H-L Xie. The Specification and Schedulability Analysis of Real-Time Systems using ACSR. In *Proc. of IEEE Real-Time Systems Symposium*, December 1995.
- [6] Uffe Engberg and Kim S. Larsen. Efficient Simplification of Bisimulation Formulas. In *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 111–132. LNCS 1019, Springer-Verlag, 1995.
- [7] Richard Gerber, Dongin Kang, Seongsoo Hong, and Manas Saksena. End-to-End Design of Real-Time Systems. In D. Mandrioli and C. Heitmeyer, editors, *Formal Methods in Real-Time Computing*. John Wiley & Sons, 1996.
- [8] Constance Heitmeyer and Dino Mandrioli. *Formal Methods for Real-Time Computing*. John Wiley and Sons, 1996.
- [9] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.
- [10] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19(20):503–581, 1994.
- [11] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *Computer Journal*, 29(5):390–395, 1986.
- [12] Mathai Joseph. *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall Intl., 1996.

- [13] Hee-Hwan Kwak, Jin-Young Choi, Insup Lee, and Anna Philippou. Symbolic weak bisimulation for value-passing calculi. Technical Report MS-CIS-98-22, University of Pennsylvania, Department of Computer and Information Science, 1998.
- [14] I. Lee, P. Brémont-Grégoire, and R. Gerber. A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems. *Proceedings of the IEEE*, pages 158–171, Jan 1994.
- [15] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, pages 2:237–250, 1982.
- [16] H. Lin. Symbolic graphs with assignment. In U.Montanari and V.Sassone, editors, *Proceedings CONCUR 96*, volume 1119 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 1996.
- [17] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multi-programming in A Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, 20(1):46 – 61, January 1973.
- [18] J. W. S. Liu and R. Ha. Efficient methods of validating timing constraints. In Sang H. Song, editor, *Advances in Real-Time Systems*, chapter 9, pages 199–233. Prentice Hall, 1995.
- [19] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [20] P. Paczkowski. Characterizing bisimilarity of value-passing parameterised processes. In *Proceedings of the Infinity Workshop on Verification of Infinite State Systems*, pages 47–55, 1996.
- [21] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.
- [22] R. Rajikumar, L. Sha, and J. Lehoczky. Real-Time Synchronization Protocols for Multiprocessors. In *Proc. of IEEE Real-Time Systems Symposium*, pages 259–272, 1989.
- [23] Julian Rathke. *Symbolic Techniques for Value-passing Calculi*. PhD thesis, University of Sussex, 1997.
- [24] Romesh Saigal. *Linear Programming : A Modern Integrated Analysis*. Kluwer Academic Publishers, 1995.
- [25] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.
- [26] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change Protocols for Priority Driven Preemptive Scheduling. *Real-Time Systems: The International Journal of Time Critical Computing Systems*, 1(3), December 1989.
- [27] O. Sokolsky, I. Lee, and H. Ben-Abdallah. Specification and analysis of real-time systems with paragon. *Annals of Software Engineering*, 1999. To appear.
- [28] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Real-Time Systems: The International Journal of Time Critical Computing Systems*, 1(1):27–60, 1989.
- [29] Jun Sun. *Fixed-priority End-to-end Scheduling in Distributed Real-time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.
- [30] K. Tindell and J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-time Systems. *Microprogramming*, 50(2):117–134, April 1994.

AUTOMATED FACTS GENERATION FROM RAW DATA: A PERSPECTIVE FROM THE ANDES PROJECT

Du Zhang, Vo Lee

Dept. of Computer Science
California State University
Sacramento, CA 95819-6021

Joseph Friedel

Navy PMO
SPL
Mt. View, CA 94039

Robert Keyser

Near Field Test Range
SM-ALC/LHHDEC
McClellan AFB, CA 95652

ABSTRACT

One of the problems in developing ANDES, an expert system for diagnosis of the US Air Force phased-array satellite ground station antenna system at the Antenna Repair Facility of McClellan Air Force Base, is how to automatically generate useful facts from raw antenna test data so as to facilitate the diagnostic process. In this paper, we discuss a statistic model for the raw test data and verify that the model is valid. Algorithms are then developed based on the model that are used to automatically generate diagnostic facts for the antenna diagnosis process. Our experience indicates that statistic models are useful in automating the knowledge acquisition process and that domain specific information is needed in defining automation algorithms.

Keywords: Phased-array antenna, diagnostic expert system, normally distributed data, automated facts generation.

1. BACKGROUND

In the early 1990's the Antenna Repair Facility (ARF) at McClellan Air Force Base received the task of maintaining an LS band satellite ground station antenna system. Each antenna in the antenna system is a three-by-four-foot flat phased-array with 128 identical elements (or subantennas) on two radio frequency (RF) circuit boards, with two low noise amplifiers (LNA), and a four-voltage type supply. Four digital circuit cards provide steering control to the 128 elements. Each element has about two dozen physical components (including a phase shift circuit consisting of RF diodes, inductors, capacitors, and resistors), thus resulting in over 2000 components in each antenna. Three diode pairs provide shifting at increments of 45 degrees for each element. The remaining major components are the antenna case and radome (Figure 1).

Though strides have been made on automating measurement collection and testing methods since ARF has been given the antenna maintenance task [2,3], diagnosis is still complex and analysis-intensive, and remains predominantly a human task. Just on the element and major component level, the input data measurements number in the thousands. The diagnostic data consist of dozens of tests. The required skill level for the repair technician/engineer is high, demanding knowledge in electronics, mechanics, RF, computer operation and data processing. Training a new technician or engineer to repair this type of antennas is lengthy and expensive. Several years of hands-on experience are usually needed before someone becomes effective and efficient in the job.

As the McClellan Air Force Base is scheduled to close in a few years, ARF faces a new challenge of maintaining their in-house expertise and production levels as manpower decreases, and transferring their many years of skill in antenna maintenance from their closing base to the acquiring base. To meet the challenge, an expert system called ANDES (Antenna Diagnostic Expert System) is being developed to help human engineers improve the antenna diagnostic process.

ANDES is developed in CLIPS, a rule-based expert system shell environment [4,9]. It is designed with a layered modular structure where each layer consists of a group of modules implementing a particular system function (refer to Figure 2). Functions provided at a lower layer can only be invoked by functions in a higher layer, but not vice versa. On the other hand, data defined in a higher layer can be exported to a lower layer module, but not vice versa. Such a design lends itself to easy system correctness verification, flexibility and expandability. Because of the nature of the antenna diagnostic process, certainty factors are incorporated into its knowledge base. ANDES has a command language interface with the users and handles most of the antenna diagnostic problems (ten different categories). The ANDES' experience indicates that during the times of downsizing, streamlining and restructuring, expert systems offer a viable and sometimes pivotal means (1) to capture and preserve the enterprise expertise from a closing base, (2) to provide a training tool for the acquiring base, and (3) to help maintain productivity during the base shutting down period.

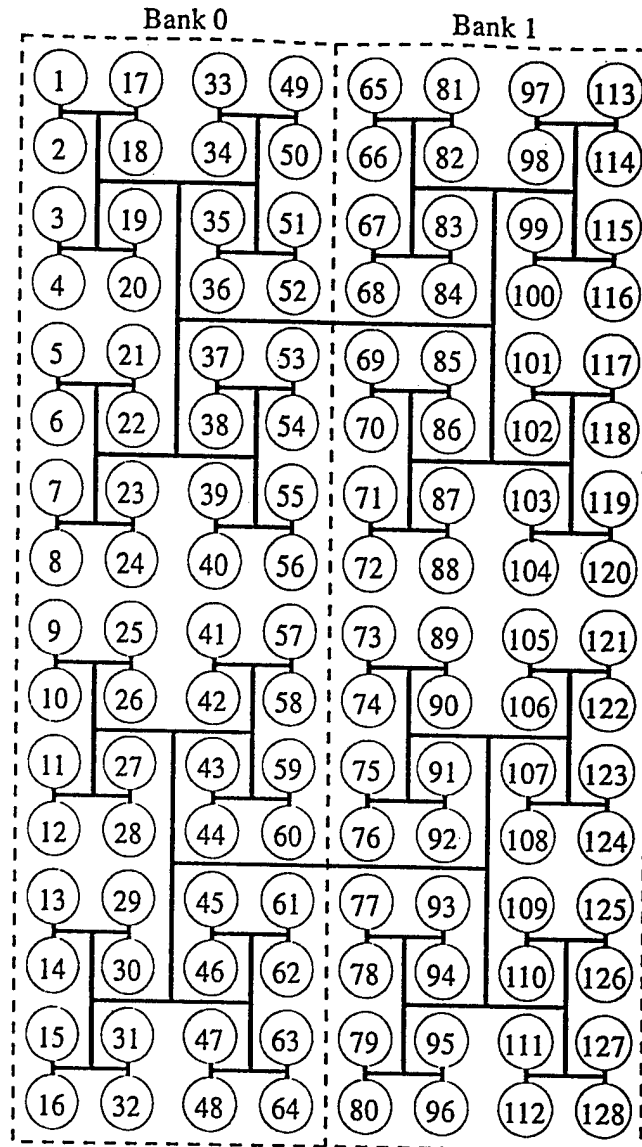


Figure 1. 128-element phased-array antenna.

One of the problems during the development of ANDES is how to automatically generate useful facts from raw antenna test data so as to facilitate the diagnostic process. Because of the lack of knowledge acquisition tools to aid the task, we define a statistic model for the raw test data and verify that the model is valid. Algorithms are then developed based on the model that are used to automatically generate diagnostic facts from the test data for antenna diagnosis process. Our experience indicates that statistic models are useful in helping automate the knowledge acquisition process and that domain specific information is needed in defining the automation algorithms.

The focus of the paper is thus on the facts generation layer in ANDES' structure (see Figure 2). The remainder of the paper is organized as follows. Section 2 deals with the raw antenna test data and how human experts perform the data to facts generation process. Section 3 discusses the statistical model for the test data. Its validity is verified in Section 4. Algorithms based on the model are given in Section 5 that are utilized by ANDES in generating diagnostic knowledge from raw antenna test data. Finally, Section 7 concludes the paper with remarks on future work.

2. ANTENNA SCAN DATA

In identifying symptoms during an antenna diagnostic process, there needs to be facts that describe an antenna's properties in terms of some qualitative terms such as low, high, good, bad, and so forth. Before we are able to discuss how

these facts are directly generated from the raw test data (RF diagnostics and back transform scan data), we need to take a look at its format as follows.

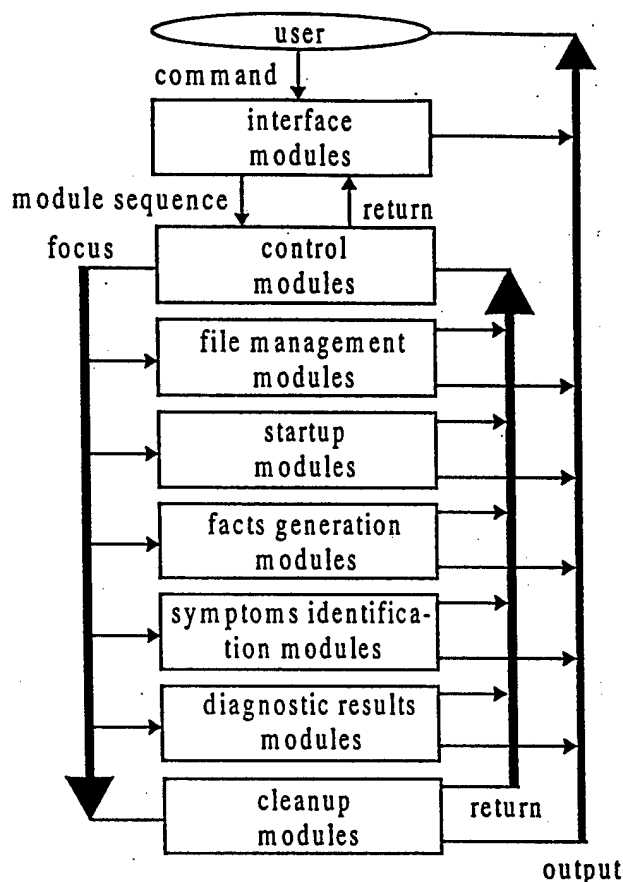


Figure 2. Structure of ANDES.

(1,1) (1,2) (1,3) (1,3) (1,4)...(1, n_x-4) (1, n_x-3) (1, n_x-2) (1, n_x-1) (1, n_x)
 (2,1) (2,2) (2,3) (2,3) (2,4)...(2, n_x-4) (2, n_x-3) (2, n_x-2) (2, n_x-1) (2, n_x)
 (3,1) (3,2) (3,3) (3,3) (3,4)...(3, n_x-4) (3, n_x-3) (3, n_x-2) (3, n_x-1) (3, n_x)
 (4,1) (4,2) (4,3) (4,3) (4,4)...(4, n_x-4) (4, n_x-3) (4, n_x-2) (4, n_x-1) (4, n_x)
 (5,1) (5,2) (5,3) (5,3) (5,4)...(5, n_x-4) (5, n_x-3) (5, n_x-2) (5, n_x-1) (5, n_x)
 (6,1) (6,2) (6,3) (6,3) (6,4)...(6, n_x-4) (6, n_x-3) (6, n_x-2) (6, n_x-1) (6, n_x)
 (7,1) (7,2) (7,3) (7,3) (7,4)...(7, n_x-4) (7, n_x-3) (7, n_x-2) (7, n_x-1) (7, n_x)
 (8,1) (8,2) (8,3) (8,3) (8,4)...(8, n_x-4) (8, n_x-3) (8, n_x-2) (8, n_x-1) (8, n_x)

There are 8 rows in a data file where each row has n_x scan readings across. Data items represent uniformly spaced readings taken on a radome antenna of Figure 1, where row 1 in the file represents scan readings taken over elements 113, ..., 128; row 2 represents scan readings taken over elements 97, ..., 112; and so on, and row 8 represents scan readings taken over elements 1, ..., 16.

2.1. Facts Generation by Human Experts

Human experts' approach to generating diagnostic facts from the RF diagnostic scan values is summarized as follows: (1) They deal with each row at a time, creating a 2-dimensional graph based on scan data from that row. (2) Knowing that if the curve is flat, then the elements within this row are good, they then look at each graph and make judgments on those few readings in this graph which are unacceptably lower or unacceptably higher than the other readings which appear relatively flat, or stable. These elements are marked as suspect. (3) The human expert then looks at suspect elements over all phase shift settings to determine the specifics of the element failure.

2.2. Postulation

If the scan data have the normal distribution or are approximately normal, then from a confidence level $1-\alpha$ (to be provided by the experts), the standard normal distribution curve, along with the scan data, can be used to estimate an interval $\langle g_{\min}, g_{\max} \rangle$ where the "good" data values lie. Any value smaller than or equal to $g_{\min}$ would be considered low, and any value greater than or equal to $g_{\max}$ would be considered high.

3. THE STATISTICAL MODEL

3.1. Empirical Rule: Standard Normal Distribution

Given a fairly large sample from a normal population where μ and σ are not known, it is not unreasonable to use the sample mean $\bar{X}$ and the sample standard deviation s as rough estimates for the population mean μ and the population standard deviation σ , respectively. The intervals have the following data coverage:

Intervals	Percentage of Data (approximate)
$\langle \bar{X} - s, \bar{X} + s \rangle$	68
$\langle \bar{X} - 2s, \bar{X} + 2s \rangle$	95
$\langle \bar{X} - 3s, \bar{X} + 3s \rangle$	99.7

Recall from statistics that for any normal variable x with mean μ and standard deviation $\sigma > 0$,

$$P(\mu - \sigma < x < \mu + \sigma) = 0.6826;$$

$$P(\mu - 2\sigma < x < \mu + 2\sigma) = 0.9544;$$

$$P(\mu - 3\sigma < x < \mu + 3\sigma) = 0.9974;$$

$$P(\mu - 4\sigma < x < \mu + 4\sigma) = 0.99994;$$

$$P(\mu - 5\sigma < x < \mu + 5\sigma) = 0.9999994; \text{ and so forth.}$$

That is, let $\lambda > 0$, then since $z = (x - \mu)/\sigma$,

$$\mu - \lambda \cdot \sigma < x < \mu + \lambda \cdot \sigma \Leftrightarrow (\mu - \lambda \cdot \sigma - \mu)/\sigma < z < (\mu + \lambda \cdot \sigma - \mu)/\sigma \Leftrightarrow -\lambda < z < \lambda;$$

and, $P(-\lambda < z < \lambda)$, which is equal to $2P(0 < z < \lambda)$, can just be obtained by summing the appropriate areas under the standard normal distribution curve. It follows that

$$P(\mu - \lambda \cdot \sigma < x < \mu + \lambda \cdot \sigma) = P(-\lambda < z < \lambda)$$

(NOTE: for a continuous random variable x , $P(a \leq x \leq b) = P(a < x < b)$, because adding a and b to the interval does not increase the area directly above the interval). Then from comparing the Empirical Rule and the results here, it can safely be said that $s = \sigma$ and $\bar{X} \approx \mu$ as n becomes large. So assuming n is large enough to make these estimations, it is permissible to use the standard normal distribution curve to estimate probabilities when s and $\bar{X}$ are known.

3.2. Interval $\langle g_{\min}, g_{\max} \rangle$

Now suppose there is a fairly large sample, the population is normal, and λ corresponds to some level of confidence $1-\alpha$ (i.e., if $\lambda = 1$, the estimated level of confidence is 0.6826) such that x is expected to be in the interval $\langle \bar{X} - \lambda \cdot s, \bar{X} + \lambda \cdot s \rangle$. Then if there is an x' , not in $\langle \bar{X} - \lambda \cdot s, \bar{X} + \lambda \cdot s \rangle$, it can be said that with an estimated confidence level of $1-\alpha$, x' is a candidate for not being a value in $\langle \bar{X} - \lambda \cdot s, \bar{X} + \lambda \cdot s \rangle$, since no x is expected to be out of $\langle \bar{X} - \lambda \cdot s, \bar{X} + \lambda \cdot s \rangle$ with an estimated confidence level of $1-\alpha$.

Thus, applying the just discussed concept to the scan values, let $\langle g_{\min}, g_{\max} \rangle = \langle \bar{Z} - \lambda \cdot s_z, \bar{Z} + \lambda \cdot s_z \rangle$, where $\bar{Z}$ is the average of the scan values and s_z is the standard deviation of the scan values, provided that the scan values are normally distributed and that there is a sufficient number of scan values in each row of scan. For the latter requirement, the requirement that $n_x \geq 30$ can be made.

3.3. Guidelines to the Selection of $1-\alpha$

Clearly λ depends on the estimated level of confidence $1-\alpha$. It is not always desirable to have $1-\alpha$ extremely close to 1, because that would increase the estimated confidence interval, making the estimation of $\bar{X}$ less accurate. Thus, there is a trade-off between the estimated level of confidence and accuracy: higher confidence implies less accuracy, and more accuracy implies less confidence. In statistics, levels of confidence often used are 80%, 90%, 95%, 98%, 99%, which approximately correspond to λ values of 1.28, 1.64, 1.96, 2.33, 2.58 respectively. Notice, here λ is just the value of z such that the area under the standard normal curve from 0 to z is $(1-\alpha)/2$ ($0 < 1-\alpha < 1$, because λ approaches infinity as $1-\alpha$ approaches 1, making λ undefined as $1-\alpha$ reaches 1. And, as $1-\alpha$ approaches 0, λ approaches 0, and $\lambda = 0$ when $1-\alpha = 0$;

however, if $\lambda = 0$, the interval $\langle \bar{X} - \lambda \cdot s, \bar{X} + \lambda \cdot s \rangle$ becomes nil, so the requirement $\lambda > 0$ is necessary to keep the interval $\langle \bar{X} - \lambda \cdot s, \bar{X} + \lambda \cdot s \rangle$, which is open at both ends, defined. Thus, this requirement implies that $1 - \alpha > 0$.

3.4. Determining λ Given the Confidence Level

λ is needed when given the confidence level in order to use $\langle g_{\min}, g_{\max} \rangle$. Recall from statistics that the normal distribution is given by the function

$$f(x) = \frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}},$$

where x is the random variable, μ is the mean and σ is the standard deviation. The function simplifies to $f(x) = \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}}$ for the standard normal distribution because of its definition ($\mu = 0$ and $\sigma = 1$). Thus,

$$P(0 < z < \lambda) = \int_0^\lambda \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}} dx.$$

This is one of those functions which cannot be integrated. Therefore, some kind of series should be used to estimate it. For simplicity, the following Simpson's Rule can be used.

$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \dots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n))$$

This means that the maximal error of using Simpson's Rule with $a = 0$ and $b = \lambda$ is $\frac{3 \cdot \lambda^5}{180n^4 \sqrt{2\pi}}$. Furthermore, since $\lambda > 0$, if n is some even constant, necessarily larger than 1, the maximal error in using Simpson's Rule in the interval $[0, c]$ is $\frac{3 \cdot c^5}{180n^4 \sqrt{2\pi}}$. Thus, if no more error than ϵ is allowed, then $\frac{3 \cdot c^5}{180n^4 \sqrt{2\pi}} < \epsilon$ or $n > \left(\frac{c^5}{60\epsilon \sqrt{2\pi}} \right)^{1/4}$.

Using the bisection method to determine a particular λ' where $P(0 < z < \lambda') = (1 - \alpha)/2$, it is found that after λ exceeds 4, with no more error than $|0.5 - 0.49997| = 0.00003$, $P(0 < z < \lambda) = 0.5$, where no more error than 0.00003 implies $P(0 < z < \lambda) - P(0 < z < 4) < 0.00003$ for $\lambda > 4$. Of course this would bring in the restriction that $0 < (1 - \alpha)/2 \leq 0.49997$, or

$0 < (1 - \alpha) \leq 0.99994$. If $c = 4$ and $\epsilon = 0.00001$, then $n > \left(\frac{c^5}{60\epsilon \sqrt{2\pi}} \right)^{1/4} \approx 28.7253$; thus, we can let $n = 30$, since n must be even.

3.5. Regression Function $\hat{Z}(x)$

Since the elements are not exactly at the points where the scan values are taken, their scan values would have to be estimated before the interval $\langle g_{\min}, g_{\max} \rangle$ can be used to determine the conditions for the elements. This would require some kind of regression $\hat{Z}(x)$. We need to rely on the scan values to find $\hat{Z}(x)$ for an element e_c of column c in matrix scan.

In the following matrix scan, the integers 1, 2, 3, ..., n_x mark the positions where scan readings are taken in the x -axis; the integers 1, 2, 3, ..., n_u mark the positions where the elements are located in the u -axis, which is in the same direction as the x -axis; the integers 1, 2, 3, ..., n_y mark the position where scan readings are taken in the y -axis; and sv marks the positions where scan readings, or values, are taken for a pair (x, y) . Notice, the positions marked by integers 0, $n_x + 1$, $n_x + 1$ and $n_y + 1$ are introduced because the scan readings and element positions are uniformly spaced on the radome antenna; that is, although these positions do not represent actual elements, they represent the fact that an element next to an edge of the antenna is within the edge the same distance it is within an adjacent element.

Then $\frac{n_u + 1}{n_x + 1} = \frac{u}{x} \Rightarrow x(u) = \frac{u \cdot (n_x + 1)}{(n_u + 1)}$, where $x(e_c)$ is the relative position of element e_c in the x-axis (Note: to get

the absolute position, ΔX is required where two successive scan readings are taken in the x-axis. Thus, the absolute position of e_c is $X(e_c) = (\Delta X)x(e_c)$. For scan reading r , column c of matrix scan, and element e_c , where $r = \{1, 2, \dots, n_y\}$ and $c = \{1, 2, 3, \dots, n_x\}$, $x(e_c - \delta) < r < x(e_c + \delta)$ can be used to determine whether or not scan reading r in column c should be used as one of the scan values to derive $\hat{Z}(x)$, which is needed to evaluate $\hat{Z}(x(e_c))$, the estimated scan value for element e_c .

[Parameters of Matrix Scan]								
0	1	2	3	4	5	...	(n_y)	($n_y + 1$) $\rightarrow y$
.	1	sv	sv	sv	sv	...	sv	
.	.	.	.	.	.	...	.	
1	.	.	.	.	.	...	.	
.	.	.	.	.	.	...	.	
.	.	.	.	.	.	...	.	
(n_u)	.	.	.	.	.	...	.	
.	.	.	.	.	.	...	.	
.	(n_x)	sv	sv	sv	sv	...	sv	
($n_u + 1$)	($n_x + 1$)							
↓	↓							
u	x							

In determining δ in general, the number of scan readings used between u_1 and u_2 ($u_1 < u_2$) in column c is given by

$$n_c(u_1, u_2) = (\lceil x(u_2) \rceil - 1) - (\lfloor x(u_1) \rfloor + 1) + 1 = \lceil x(u_2) \rceil - \lfloor x(u_1) \rfloor - 1.$$

So the total number of scan readings used from column c is $(n_c)_{\text{tot}} = \sum_{i=1}^{n_u} n_c(i - \delta, i + \delta)$. Because the scan data are collected

based on $n_x = 211$, $n_u = 16$ and $n_y = 8$, the behaviors of δ indicate that when $\delta = \frac{1}{2}$ it reflects a particular element as much as possible without a lot of interference from the readings of nearby elements.

Having chosen δ , the regression function to fit the scan readings such that $x(e_c - \delta) < r < x(e_c + \delta)$ can be determined next. The goal is to define, for a particular column, c , of scan values with variables x and z (x is the dependent variable and z is the independent variable), a regression function $\hat{Z}(x)$ to estimate for an element e_c a scan value $\hat{Z}(x(e_c))$. Since the overall goal is to find a flat curve, the best fit line with slope zero for the scan readings should be a good candidate as the regression function, meaning that $\hat{Z}(x)$ is just the average of the scan readings in column c where $x(e_c - \delta) < r < x(e_c + \delta)$. Thus, if δ is such that all scan values in column c are used, then $\hat{Z}(x(e_c)) = \bar{Z}$ for any e_c .

4. MODEL VERIFICATION

To verify that the model is valid and the antenna scan data do follow the normal distribution, twelve randomly picked non-corrupted complete sets of scan data are obtained from a collection of about sixty data sets. Each set of data contains files for RF diagnostic scans of both the magnitude and the phase types at phase shifts of 0, 45, 90 and 180 degrees. Typically RF scan data for a phase shift of 315 degree are not collected, so sets not containing RF scan for 315 degree are not considered incomplete. However, if scan data for 315 degree are included in a set, then there must be a file for type magnitude and a file for type phase in order for this set to be considered complete. The twelve sets of data selected are from twelve different antennas, respectively.

As a first step, a pair of RF scan magnitude and RF scan phase files at some phase shift is randomly selected from the twelve data sets for an initial probe. Each of the two files has 211 scan values for each of the eight rows, making up 1,688 scan values for a file. Figure 3 and Figure 4 compare the actuals and estimates of the scan data in these two files.

Given n values, a sample mean $\bar{X}$ and a sample standard deviation s are calculated. Let

$\text{count} = \text{the number of values in } < \bar{X} - \lambda \cdot s, \bar{X} + \lambda \cdot s >$

$\text{actual} = P(\bar{X} - \lambda \cdot s < x < \bar{X} + \lambda \cdot s) = \text{count}/n,$

$\text{estimate} = P(-\lambda < z < \lambda),$

$P(-\lambda < z < \lambda)$ can be obtained by finding the area under the standard normal distribution, the normal distribution curve where $\mu = 0$ and $\sigma = 1$. By comparing *actual* and *estimate*, an idea of whether or not x is normal will be observed. If *actual* = *estimate* for all λ s, then x is normal. If *actual* $\approx$ *estimate* for all λ s, then x is approximately normal.

Let λ be in the $[0.1, 5.0]$ with $\Delta\lambda = 0.1$, for the scan values in each row of a particular file, λ versus *actual* and λ versus *estimate* are graphed. Also letting $\text{error} = \text{actual} - \text{estimate}$, the average error and the standard deviation of the errors are calculated for each graph. Since *estimate* is 0.9996, 0.99994, 0.999994, and 0.9999994 for $\lambda = 3.5, 4.0, 4.5$ and 5.0 , respectively, and *estimate* = 1 for $\lambda > 5$, the range of λ s should be sufficient in seeing if the data exhibit a normal distribution. The tool used to generate these graphs and calculate these figures is Mathematica [8].

Figure 3: Actuals and estimates for all scan values of RF scan of type magnitude, phase shift of 0 degree (avg. error= 0.0111; std. dev. of errors= 0.0026) (dashed curve corresponds to actuals, and solid curve corresponds to estimates)

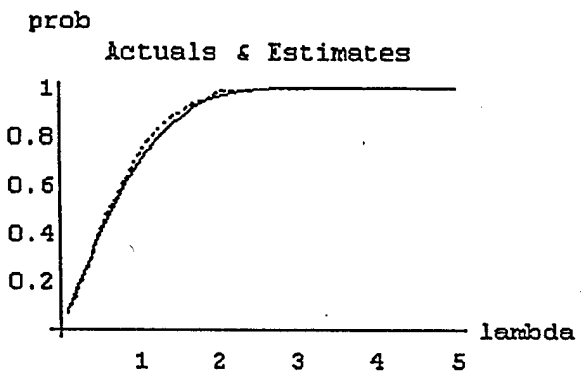
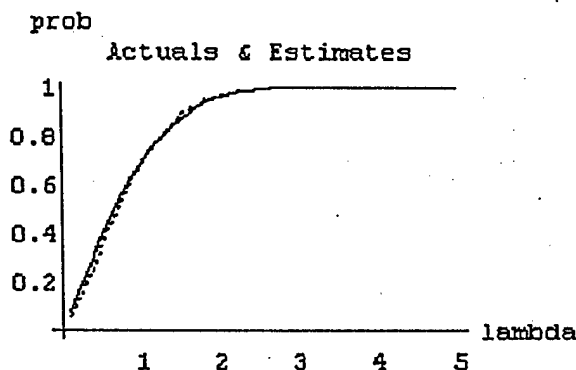


Figure 4: Actuals and estimates for all scan values of RF scan of type phase, 0 degree (avg. error= 0.0073; std. dev. of errors= 0.0025) (dashed curve corresponds to actuals, and solid curve corresponds to estimates)



From Figures 3 and 4 the randomly selected pair of files contains data that appear to have the normal distribution, so normality does seem to exist. Then, next we look at all the files in the twelve sets of data to see if the data in each file exhibit the similar characteristics. Instead of generating graphs for the data in every file of the twelve data sets, we choose, as the average actual, the average of the actuals for each λ , where $\lambda \in [0.1, \text{to } 5.0]$ with $\Delta\lambda = 0.1$, of all rows from all files for a particular type (either magnitude or phase), of a particular degree, (either 0, 45, 90, 180, or 315) in the twelve sets of data. Thus, if the "average" row appears to be approximately normal, then overall it can be considered that the RF scan data of a particular type and degree are approximately normal. Taking this approach, graphs are generated for the 10 possible "average" rows; that is, five for rows of type magnitude, with phase shifts of 0, 45, 90, 180 and 315 degrees, and five for rows of type phase with phase shifts of 0, 45, 90, 180 and 315 degrees. In addition, the standard deviation of the average actuals versus λ is plotted for each of the ten cases, to show the dispersion of the actuals at each λ . The results show that the scan data do observe the normal distribution, even though we only include the graphs for two cases due to space limit (refer to Figures 5-8).

Looking at the figures of the standard deviations and the averages of the errors, it can be said that overall the scan values are pretty close to being normally distributed. In addition, the plots of the standard deviations of the actuals are not totally surprising. For example, when λ is small, since the actuals are small, due to the small intervals used to obtain the actuals, the standard deviation of the actuals is expected to be small; when λ is large enough, the standard deviation of the actuals is expected to become smaller only for larger λ , since large intervals are expected to cause more consistent actuals. In any rate, prior to becoming large enough as λ increases, the standard deviation of the actuals at λ is expected to increase, since the intervals used to obtain the actuals are not large enough to make the actuals consistent but yet the intervals are becoming larger, giving the actuals more freedom to take on different values.

Figure 5. Average actuals and estimates
for RF scan of type magnitude, degree 0
(avg. error= 0.0205; std. dev. of errors= 0.0136)
(dashed curve corresponds to actuals, and solid
curve corresponds to estimates)

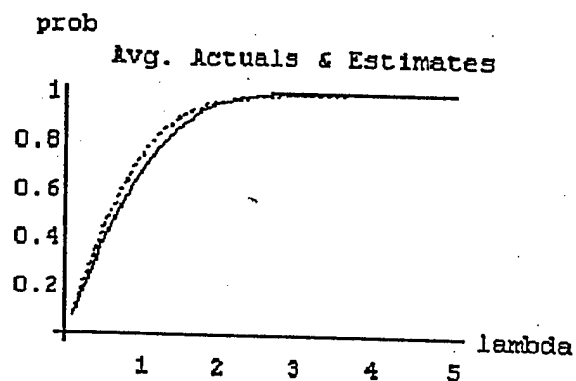


Figure 6. Standard deviation of average
actuals for RF scan of type
magnitude, degree 0.

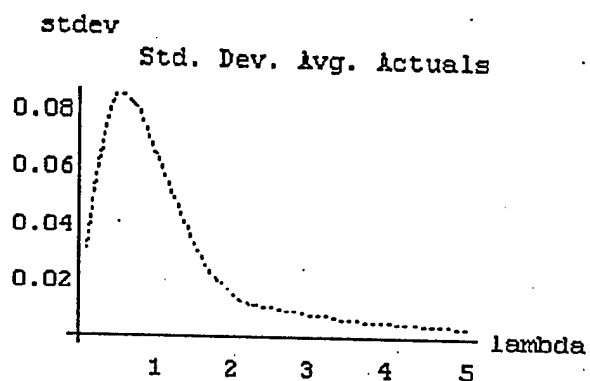


Figure 7. Average actuals and estimates
for RF scan of type phase, degree 90
(avg. error= 0.0043; std. dev. of errors= 0.0027)
(dashed curve corresponds to actuals, and solid
curve corresponds to estimates)

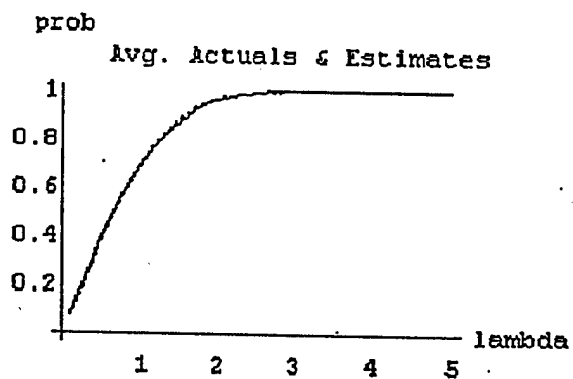
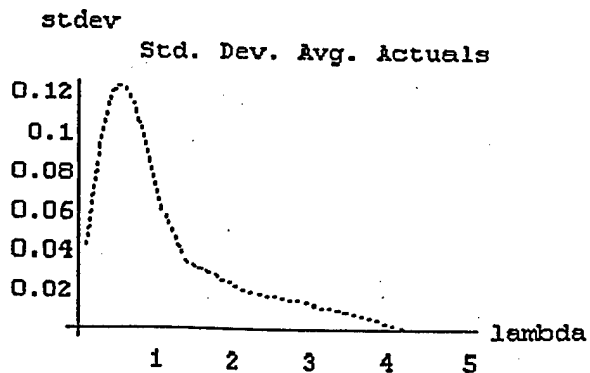


Figure 8. Standard deviation of average
actuals for RF scan of type phase, degree 90.



5. ALGORITHMS FOR AUTOMATIC FACTS GENERATION

/* δ , n_x and n_u are assumed to be global constants. */

function $x(u)$ *return* ($\frac{u \cdot (n_x + 1)}{(n_u + 1)}$);

function $n_c(u_1, u_2)$ *return* ($\lceil x(u_2) \rceil - \lfloor x(u_1) \rfloor - 1$);

function $\text{find_elmt_cond}(1-\alpha, \text{scan})$

/* scan is a n_x -by- n_y matrix containing all the scans readings. */

/* zecavg is a n_u -vector used to ultimately store the averages of the z where $(x(e_c) - \delta) < r < (x(e_c) + \delta)$. */

/* sumzc and sumzczc are maintained to calculate the average z of column c , zecavg , and the sample standard deviation of column c , sc , which are needed to determine if e_c is low, good or high. */

for ($c = 1$ to n_y) *do*

 initialize all matrix elements of zecavg to be 0;

 initialize $\text{sumz} = 0$, $\text{sumz2} = 0$

for ($r = 1$ to n_x) *do*

if $x(0.5) < r < x(1.5)$

$\text{zecavg}[1] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(1.5) < r < x(2.5)$

$\text{zecavg}[2] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(2.5) < r < x(3.5)$

$\text{zecavg}[3] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(3.5) < r < x(4.5)$

$\text{zecavg}[4] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(4.5) < r < x(5.5)$

$\text{zecavg}[5] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(5.5) < r < x(6.5)$

$\text{zecavg}[6] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(6.5) < r < x(7.5)$

$\text{zecavg}[7] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(7.5) < r < x(8.5)$

$\text{zecavg}[8] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(8.5) < r < x(9.5)$

$\text{zecavg}[9] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(9.5) < r < x(10.5)$

$\text{zecavg}[10] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(10.5) < r < x(11.5)$

$\text{zecavg}[11] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(11.5) < r < x(12.5)$

$\text{zecavg}[12] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(12.5) < r < x(13.5)$

$\text{zecavg}[13] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(13.5) < r < x(14.5)$

$\text{zecavg}[14] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(14.5) < r < x(15.5)$

$\text{zecavg}[15] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

else if $x(15.5) < r < x(16.5)$

$\text{zecavg}[16] += \text{scan}[r][c]$;

$\text{sumzc} += (\text{scan}[r][c])$

$\text{sumzczc} += (\text{scan}[r][c])^2$

for $e_c = 1$ to n_u *do*

$\text{zecavg}[e_c] = n_c(e_c - \delta, e_c + \delta)$;

/* Note: scan values not used to estimate any antenna element are not used in computing sumzc and sumzczc . */

$\text{zecavg} = \text{sumzc} / (n_c)_{\text{tot}}$;

$$\text{sc} = \sqrt{\frac{(n_c)_{\text{tot}} \cdot \text{sumzczc} - \text{sumzc}^2}{(n_c)_{\text{tot}} ((n_c)_{\text{tot}} - 1)}}$$

for $e_c = 1$ to n_u *do*

if ($\text{zecavg}[e_c] \leq \text{zecavg} - \text{sc} \cdot \text{find-}\lambda(1-\alpha)$) *then*

 assert antenna element $[e_c + (c-1) \cdot n_u]$ as low;

else if ($\text{zecavg}[e_c] \geq \text{zecavg} + \text{sc} \cdot \text{find-}\lambda(1-\alpha)$)

then

 assert antenna element $[e_c + (c-1) \cdot n_u]$ as high;

else

 assert antenna element $[e_c + (c-1) \cdot n_u]$ as good;

6. CONCLUSION

In this paper, we describe how to automate the facts generation from raw test data in the development of an expert system ANDES for the phased-array antenna diagnosis. We define a statistical model for the raw antenna scan data, and then verify the model's validity in terms of the test data. Algorithms based on the model are developed for automated generation of diagnostic knowledge from raw antenna test data. ANDES accomplishes our goals of capturing and preserving the antenna diagnostic expertise, helping maintain productivity during the base shutting down period, and offering a training tool for the acquiring base. Its performance is comparable to that of a human expert.

Future work includes: further testing and evaluating, followed by refining and augmenting ANDES' knowledge base. Integration of ANDES into the diagnostic environment of the acquiring base is another highly desirable goal that will ultimately demonstrate ANDES' payoff.

REFERENCES

- [1] Chase, W. and Brown, F. *General Statistics*, John Wiley & Sons, Inc., 1992.
- [2] Friedel, J.E., Keyser, R.B. and Johnson, R.E. "Interpretation of Near-Field Data for a Phased Array Antenna," *AMTA Symposium Proceedings*, pp. 42-47, October 1993.
- [3] Friedel, J.E. and Luong, H. "Conversion of a Sonar Tank facility to a Near-Field Scanner," *AMTA Symposium Proceedings*, pp. 14-10 to 14-15, October 1992.
- [4] Giarratano, J.C. and Riley, G.; *Expert Systems*, PWS Publishing Company, Boston, MA, 1998.
- [5] Hayes-Roth, F.; Jacobstein, N.; "The State of Knowledge-based Systems," *Communications of ACM*, Vol. 37, No. 3, pp. 27-39, 1994.
- [6] Lee, V. ANDES: A Phased-Array Antenna Diagnostic Expert System, MS degree thesis, Department of Computer Science, California State University, Sacramento, Summer 1998.
- [7] *Mathcad 5.0 Help Files*, MathSoft, Inc., 1986-1994.
- [8] *Mathematica 2.2 Help Files*, Wolfram Research, Inc., 1993.
- [9] Software Technology Branch, Lyndon B. Johnson Space Center; *CLIPS Reference Manual*, Vol. 1-3, 1993.
- [10] Steward, J.; *Calculus*, Wadworth, Inc., Belmont, CA, 1987.
- [11] Zhang, D. Friedel, J.E., Lee, V., Keyser, R.B. and Kho, J.W. "ANDES: A Phased-Array Antenna Diagnostic Expert System," *Proceedings of the Fourth World Congress on Expert Systems*, pp.286-293, March 1998.

Pitfalls of Formality in Early System Design

David Robertson

Division of Informatics, University of Edinburgh, Edinburgh, Scotland
Tel: +131-650-2709, Fax: +131-650-6516, Email: D.Robertson@ed.ac.uk

Abstract

One of the advantages of using formal methods in design should be that we can be precise about where our methods fail. However, it is rare to find discussions in the literature of problems in applying formal methods - particularly in the early stages of design. One reason for this is that failures are often caused by the context in which a method is applied, rather than by some purely technical limitation. Using examples from research in which I have been involved I shall describe some of the pitfalls I have encountered and which I have observed frequently in the research of others.

1 Introduction

Berry, in these proceedings [1], advocates the use of appropriately chosen and applied formal methods in the early stages of design lifecycles - observing factors outside the technical considerations of the methods themselves which influence their success. This paper is complementary to Berry's discussion because it looks at some factors which can lead to failure. It has been unfashionable for those using logic-based methods to introspect about causes of failures. This is a pity because without the possibility of failure our research isn't experimental and without the ability to learn from failure we are unlikely to develop robust engineering methods.

To avoid being accused of preying on other unsuspecting researchers, I have taken all the examples of pitfalls from my own research and where others have been involved I have generalised the examples. In none of the projects I describe were the mistakes fatal but this was largely because they were small scale, which made corrections simpler to make. Larger research efforts might not have the manoeuvrability to fix these problems at acceptable cost. Once explained, some of the pitfalls may seem obvious but, given how frequently signs of these can be found in the research of others, I suspect that they are much easier to recognise with hindsight than to predict.

In what follows, I use "model" to refer to the set of logical expressions used to describe some problem. This is used instead of the word "specification" because in early design we often use models of problems which are not directly specifications of systems (see [11] or the introduction of [12]). I have also used the phrase "inference system" in a broad sense to denote any system which has been built with the intention of using or synthesising expressions in a mathematical logic.

2 Choice of Inference System

Formal definition takes place within a chosen system of inference. Although in theory there is great deal of overlap between inference systems, so that it should be easy to translate definitions from one to another, in practice our choice of system emphasises particular features of the problem. For example, a modal temporal logic is a natural choice if we are tackling a problem where we need to prove that certain propositions eventually hold of our system model. However, those inference rules which allow us to deal readily with the temporal aspects of problems may be a distraction in problems where we don't need to prove temporal properties. This issue is well known and is being addressed, in part, by those whose interest is in making it easier to combine or translate between inference systems. There remain some subtle difficulties which cannot be solved simply by deepening formal theory.

There are hundreds of specialised formal calculi, for describing certain forms of uncertainty; for expressing temporal information; etc. If we are lucky enough to have a problem which obviously suits a particular class of calculi then we may reduce our choice to a few "front runners" but we may have to dig deeply into the theory of each competitor before we can decide on a winner. For example, in [5] it is demonstrated that any problem representable in Dempster-Shafer theory can be represented in the Incidence Calculus and *vice versa*. Unwary readers might be led to believe by this result that the two calculi are "equivalent". This would be a mistake because in [10] a version of Incidence Calculus is described which for some situations involving dependent evidence gives the intuitively correct answer when Dempster-Shafer theory does not. In other words, the comparison between these two inference systems depends on

exactly which version of each system we are comparing and on what sort of equivalence we wish to demonstrate. It is hard to understand such comparisons deeply without becoming an expert in all of the inference systems one might want to use - a prohibitively time consuming task since domain problems do not always neatly fit into a small class of related inference systems - but without deep knowledge we can easily misjudge the capabilities of the systems we intend to use.

One reaction to the problem of selecting a specialised logic is to begin projects with a general-purpose logic, perhaps with a commonly accepted computational interpretation. For instance, we might use Horn clauses with the resolution-based inference strategy familiar to logic programmers. We would then write all our formal expressions in the form : $C \leftarrow P_1 \wedge \dots \wedge P_n$, where C is the (atomic) conclusion and each P is an atomic condition with the special condition, *true*, being used in cases where C is true unconditionally. What happens if we find that we need to represent concepts such as temporal change within our model? There are at least two technical solutions:

- Use the same inference system but add axioms which deal with temporal effects. An example is the Event Calculus [8].
- Use a different inference system containing proof rules which deal with temporal change. An example is the Temporal Logic of Actions [9].

Whichever solution is chosen, there is a cost to the extension which is not purely technical: it is necessary to negotiate the extension to the logic with others working on the project and to re-train them in the tasks which they must now perform in the new style. The reason why this can become a serious problem in early design is that the problems we are modelling often are loosely bounded and it is easy to find complex logical puzzles within them. Why stop at a temporal extension when there are also interesting issues of typing, higher-order function application, *etc* which we can find in the problem if we stare hard enough? Such problems fascinate the logicians in a project so it is easy to slide down a "slippery slope" of increasing language complexity, incurring a cumulative cost in negotiation and education in addition to the normal technical overheads.

3 Boundaries of Formality

Most uses of formality in early design require some degree of interaction with humans in a domain of application. They may be manipulating formal expressions in order to describe a model or assisting in the knowledge acquisition needed to build a domain-specific synthesis system or analysing information deduced from a model during validation. Whatever the point of interaction and regardless of the sophistication of the interface to the formal system, it will be necessary to commit to a human interpretation of at least some of the formal symbols manipulated by the inference system. A system for deciding on the mappings between human and formal language in some domain is often called an ontology. To work perfectly, everyone who needs to use an ontology must follow the same conventions in relating human to formal representation. There are two surprises here: the first is how deep differences in interpretation of ontologies can be; the second is that usable systems can be produced even when such differences occur.

In the knowledge based systems community, ontological research has become a major research theme (see [14] for a survey). The most common strategy is to fix on a bounded domain of application and devise a restricted formal language which those working in the domain use to describe problems. This shared language is used as the basis for sharing information. In our ecological work [13] we attempted to do this for a class of ecological modelling problems - the idea being to use the domain ontology to provide a language in which ecologists describe problems and then to use the formal problem description to control the generation of appropriate ecosystem models. This meant that we had to worry about what words like "biomass" should be taken to mean. Depending on who one speaks to, the answer to this question is different. The most crude definition is "mass of biological material". A more precise definition might be "mass of biological material once all water has been removed". For specialist sub-domains the definition might be "mass of biological material which has been subjected to the following treatment to remove water...". In other words, there is no broadly applicable definition of basic concepts like "biomass" which suits all situations. The search for consensus on such issues quickly took us into waters which were uncharted even for our domain experts. One reaction to this might be to shy away from domain-specific ontologies and rely instead on "domain-independent" terminology from stable technical communities. This route also fails, as we explain below.

An example of a stable technical community is in uncertain reasoning but even here we can find differences in formal interpretation of basic expressions. A manifestation of this appeared in an electronic mail debate which took

place on the uai mailing list in the early summer of 1998. At issue was the common practice of referring to the X in the expression $P(X = x)$, where P is the probability that X has value x , as a "random variable". This sort of notation is often used formally to represent statements like "The probability that the colour of my car is black", which might be $P(car_colour = black)$. The difficulty is that from the point of view of classical first-order logic it is hard to think of car_colour as a logical variable - one feels obliged to think of it as a function from cars to colours. On the other hand, it is often natural to describe it as a "variable" because it is one of the points in a problem description for which we are interested in variation. This is a different notion of variable than the classical logical one but is no less valid, and has similarities to the use of expressions such as "state variable" in process modelling. Although the use made of variables (of whatever kind) is precise and internally consistent within inference systems, the way we use natural language to refer to them differs across inference systems.

It seems that, no matter what we do, we cannot achieve a perfect ontology. Then why does formal modelling ever succeed? The reason is that we try to deploy these models in situations where the inevitable ontological mismatches will either be checked or will have negligible impact on the task which we are interested in performing. In the ecological modelling work we were careful to avoid building into the model generation mechanism any heuristics which relied on a particular interpretation of "biomass" other than as the name of an attribute of certain objects in our problem description. The price we paid for this was that our generator couldn't provide as much automatic control over model construction as it might if it made more commitments to the meaning of domain-specific concepts. We gladly paid this price in order to avoid the project foundering on arguments about the domain theory which (via the ontology) we would have been forced to embed within our inference system.

4 Fitting in with People in the Domain

A well known problem in early design is in defining who will be expected to maintain and benefit from the inference system we build. For the purposes of this paper I shall ignore that problem - although it is embarrassing to remember how frequently I and others have said "the user" in technical papers instead of some more enlightening description of the people expected to benefit from some applied system. There are other less obvious pitfalls which await the unwary.

One of the most difficult human factors to control in a research project of more than a few months duration is when to acquaint collaborators in the domain of application with the formal methods we are using. It has been said (in [4] for example) that the delay in seeing a return for an investment in formal methods is one of the key impediments to their success in industry. I suspect this is true in applications where the problem is clearly identified and the task is to describe it as succinctly and precisely as possible. In this situation we have to wait until the end before we have the "complete" result which is required. However, in early design we are much more likely to be building prototype systems to explore what the tractable problems are. We then have to choose how soon we want to have these ready. Simple prototypes can occasionally be produced extremely rapidly. The fastest I have ever done this is producing a 16-predicate logic program to demonstrate inference of chemical pathways within two hours of meeting a group of plant microbiologists for the first time. The longest time to release a prototype is roughly two years for a system we built to relate codes of practice to a safety shutdown system built using parameterisable components (see [6] for an overview of this).

Taking too long to release a prototype can cause create unpredictable problems because in that length of time the circumstances of industrial partners can change. For example, it would have been better (with hindsight) to have produced more quickly a less impressive prototype system for the safety shutdown domain because at the two year point our industrial collaborators happened to be under greater workload than hitherto, so they had less time to spend with us. Longer gestation periods give more opportunities for accidents like this to happen.

On the other hand, very rapid prototyping can raise expectations too high. Often complicated problems contain a sub-problem which is easy to tackle in an appropriate inference system. To those with little experience of such methods the initial results can seem almost magical and it can be difficult to explain that other essential tasks may be orders of magnitude more difficult. Without careful management, a fast return for investment can be as damaging as a slow return.

Ironically, rapid prototyping can also have an opposing effect. Prototypes invite (constructive) criticism and domain experts are normally good at spotting what they don't like or would wish extended. As we saw in Section 2, it is easy to feel the urge to increase the complexity of an inference system and the feedback from early prototypes may increase this pressure. Changes in support systems (such as visual interfaces) do not necessarily change in harmony

with the core inference methods so relatively small changes in the style of inference may require radical overhaul of the prototype implementation (and *vice versa*). This can tip a project into a cycle of rapid prototyping which takes a long time to achieve consensus because there is always some new and exciting variant to build.

5 Education

As Berry points out in these proceedings [1], one aim in early design is to explore the portions of the domain problem which developers don't know and which customers don't know. Normally some form of education is required on each side in order to reach a shared view of the problem which is adequate for the task in hand. There are many pitfalls in education but here I shall take one example of each type - first in the education of customers and then in the education of developers.

A popular way to make inference systems accessible to non-logicians is by providing an environment which helps those non-logicians understand the system by connecting it to concepts in which they may easily be trained. An example is techniques editing (attributable, among others, to [7] with a survey of applications in [2]) which gives an account of the structure of logic programs in terms of "conceptual structures" corresponding to tasks such as term decomposition in various forms via recursion; term construction and so on. These structural patterns apply across all the clauses in predicate and follow argument position, so predicates can be defined argument by argument according to conceptual structure rather than building clause by clause or in some more serendipitous way. Structure editors have been built which give libraries of patterns and take care of many of the details of applying them. Conveniently, these patterns also correspond to the way predicates are described when teaching logic programming: we explain that a given predicate "decomposes a term in its first argument and constructs a term in its second argument" so it is tempting to think that tools like this help non-logicians learn how to write logic programs. A group of psychologists (then at the University of Loughborough) led by Tom Ormerod ran some tests comparing the performance of undergraduate students using one of our techniques editors within a Prolog programming course to a similar group of students taught using a normal text editor. Those using the techniques editor did indeed write appropriate definitions faster than those without. However, their innate ability to understand example problems in logic programming terms did not seem to be any better than those who hadn't used the editor. One explanation for this is that they had learned to use the tool to build solutions quickly but hadn't learned to think like skilled logic programmers. Sometimes such skills aren't necessary - we may want the inference system to be a mystery to our customers because it would merely distract them. The pitfall here is in thinking that education necessarily comes with tool support.

A more pernicious (and I suspect prevalent) pitfall occurs when we consider how the developers of inference systems should be educated. It is often falsely assumed that for those expert in an appropriate logic the only additional training is in the domain itself - if our domain of application is in potato crop modelling then we need only to talk with potato crop modellers and read a few books on the subject. In fact, there is often need for additional training of the logic experts in knowledge representation. This is because much of the teaching of logic focuses on the semantics of the chosen logic and its proof theory. For these it suffices to use abstract descriptions such as $P \rightarrow \neg Q$. In fact it is often better to use these because it makes abstract notions easier to see, for example that the previous expression is equivalent in classical logic to $\neg P \vee \neg Q$. However, this sort of expertise is different from the expertise necessary to decide on an ontology for a domain and apply that ontology in a way which provides elegant, tractable descriptions of problems. Both forms of expertise are necessary to tackle problems but not all experienced logicians are good at knowledge representation. An example appears in Berry's article in these proceedings. Notice that the sort of expertise in which logicians are often deficient isn't the sort which is taught simply by presenting abstract logic differently - for instance by instantiating the logical implication above to *penguin* $\rightarrow$ $\neg$ *flies*. These are additional skills, such as the ability to choose good idealisations of problems, which are not guaranteed by an aptitude for abstract logic.

6 Evaluation

In research projects involving the construction of experimental, applied inference systems we need some form of evaluation to assess what sorts of applied problems can be tackled with the system and (if we are making usability claims) how easily it can be used by those who work in the domain. We must then worry about the cost of the evaluation effort; the degree to which empirical evaluation is likely to yield meaningful results; and, in the most

extreme case, whether it is possible for our system to fail (that is, whether we are doing an experiment at all). In each of these three cases the fact that we are evaluating an inference system can raise special difficulties.

Empirical evaluation is normally costly for any software implementation but inference systems are often particularly expensive to evaluate because, even if we do our best to fit them into standard work practices, they give people new ways of doing their jobs. Describing ecological systems in a domain-specific formal language [13, 3] was new to those who tested our model generation systems. Describing shutdown systems using parameterisable components in the way described in [6] is new to most safety engineers. This makes it difficult to relate old to new work practices. If the inference system has not been embedded carefully within its host organisation then it may be rated poorly just because it was badly introduced. If it is cosseted too carefully during field trials then its rating may be artificially high. Attempting to get this right takes time and money. For example, in the evaluations mentioned in Section 5 we were interested in comparing student performance in Prolog programming with and without a techniques editor. Both groups of students (with or without the editor) needed to be taught Prolog. To avoid artificially boosting the editor's performance because it was being introduced by those who built it, the Prolog courses and accompanying testing had to be done by other researchers (in another University). To avoid a misfit between the editor and the Prolog training course it was necessary to construct a variant of the original course into which the editor was dovetailed. To allow comparison between techniques editor and text editor courses some retrospective adaptation of the original course was needed. All of this takes considerable effort beyond that of the evaluation experiments themselves, and our example concerns an inference system which was built in order to be tested this way. Frequently, the costs of carefully controlled evaluation can be much higher.

Given the difficulty and cost of controlled evaluation it is little wonder that there are few (if any) extensive usability evaluations of larger inference systems. An alternative is to identify parts of the system and evaluate those. Here we meet at least another two pitfalls for those using logical inference systems. It is standard practice to develop these systems in a modular way so that the (internal) inference mechanisms are separable from but interacting with the (external) user interface. This can make it difficult to judge whether some faults turned up by evaluation could easily have been corrected by some adjustment to either (or both) parts of the system. Perhaps a more serious pitfall is in assuming that evaluation is compositional, in the sense that we can evaluate part of a system then combine that with evaluations of other parts to form a broader evaluation. This is seldom possible, even if our inference system is compositional, for two reasons. First, if the separately evaluated components have user interfaces then we must ensure that a good interface for one is consistent with a good interface in the other which may not be the case if they require modes of communication which are mutually antagonistic. For example, we might have a good usability evaluation for a sub-system which displays a graphical proof tree and, separately, a good evaluation of a sub-system which displays structured terms in a similar graphical style but when these are combined we get a poor usability evaluation because people are confused by the uniform visual representation of different formal concepts. The second form of compositionality problem is created by changes in demand for the system as we consider the broader lifecycle of which it must be a part. For instance, we are reasonably confident from our evaluations of novice programmers that techniques editors fit well into Prolog training courses. However, this does not mean that eventually techniques editors will be part of all introductory Prolog courses because for that to happen it would be necessary for them to mesh with the other tools which more experienced Prolog programmers want to use. Our limited evaluation says nothing about that.

Ironically, the success of logicians in producing expressive and internally consistent formal languages makes it easy for engineers to fall into the trap of designing "evaluation experiments" which are not experiments at all because there is no possibility of failure. Some examples of questions for which the answer will almost certainly be "yes", given enough effort and an expressive formal language are:

- Can a problem tackled in system *X* be tackled by a (similar) system *Y*? The answer will be "yes" if we work hard enough with system *Y*. This question is rather like comparing programming languages - only interesting if we can compare the degree of difficulty on clearly defined problems.
- Can system *Y* be made to work better than system *X* on a given problem? The answer will be "yes" if we are allowed unlimited adaptation of system *Y*. This is interesting only if the changes to *Y* are carefully constrained.
- Can people be trained to use system *X*? They almost certainly can if we choose them carefully and give them enough resource and incentives. The interesting question is whether the people, resources and incentives are available in any real domain.

- Will people become better at solving task T using system X ? If task T is of interest and system X is relevant and people use it then they are very likely to get better at the task just because they are getting practice (see Berry's comments on the "second time" phenomenon in this proceedings). The interesting question is whether they get better faster than they would have done by normal means.

None of the above are necessarily the wrong thing to do. The pitfall here is in thinking of these as giving some useful measure of empirical fitness of the method without controlling the conditions under which the measure was obtained.

7 Conclusions

Mathematical method is a prerequisite to precise experimental method. However, the former does not guarantee the latter. In preceding sections I have given examples of mistakes which mathematical method alone cannot help us to avoid. These are summarised below:

- The choice of inference system often changes during a project and this requires either alteration of the current inference machinery or its substitution by a new system. It can be difficult to make the right choice of specialist inference system and, even if it is the right choice, there can be high cost in re-education of co-researchers which accompanies each change. Too many such changes cause failure to a project because of the cumulative cost.
- The cost of producing an ontology is not just in inventing the domain-specific formal language but in maintaining it once the system is deployed, since perfect ontologies cannot be guaranteed. Over-commitment to perfecting an ontology causes failure either during development (through irreconcilable arguments over what the ontology should be) or after deployment (through inappropriate human interpretation of inference system inputs or outputs).
- Formal methods are often criticised because they take too long to yield results but this isn't necessarily true in early design. Ironically, problems such as inflated expectations and perpetual prototyping can be caused by the ability of some inference systems to tackle isolated parts of problems rapidly.
- Education is required to bring logicians and domain specialists closer together. We sometimes assume that the tools we produce will help educate domain specialists and that training in abstract logic is enough formal preparation for tackling problem domains. Both assumptions can be false.
- Through force of circumstance or naivete we may under-evaluate our systems, because we can't afford the cost; or we lack a framework for structuring the evaluation; or because our programme of research was not in essence experimental.

References

- [1] D. Berry. Formal methods: The very idea, some thoughts about why they work when they work. In *Proceedings of the 1998 ARO/NSF Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development*, Carmel, California, 1998.
- [2] A. W. Bowles, D. Robertson, W. W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying Prolog Programming Techniques. *International Journal of Human-Computer Studies*, 41(3):329-350, September 1994. Also as Research Paper 641, Dept of Artificial Intelligence, University of Edinburgh.
- [3] A Castro. *A Techniques-based Framework for Domain-specific Synthesis of Simulation Models*. PhD thesis, University of Edinburgh, 1998.
- [4] G. Cleland and D. MacKenzie. Inhibiting factors, market structure and the industrial uptake of formal methods. In *Workshop on Industrial-Strength Formal Specification Techniques*, pages 46-60, Boca Raton, Florida, 1995.

- [5] F. Correa da Silva and A. Bundy. On some equivalence relations between Incidence Calculus and Dempster-Shafer theory of evidence. In *Proceedings of the 6th Conference on Uncertainty in Artificial Intelligence*, 1990.
- [6] J. Hesketh, D. Robertson, N. Fuchs, and A. Bundy. Lightweight formalisation in support of requirements engineering. *Journal of Automated Software Engineering*, 5(2):183-210, 1998.
- [7] M. Kirschenbaum, A. Lakhotia, and L.S. Sterling. Skeletons and techniques for Prolog programming. Tr 89-170, Case Western Reserve University, 1989.
- [8] R.A. Kowalski and F. Sadri. Reconciling the situation calculus and event calculus. *Journal of Logic Programming*, 31:39-58, 1997.
- [9] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872-923, 1994.
- [10] W. Liu and A. Bundy. A comprehensive comparison between generalised incidence calculus and the dempster-shafer theory of evidence. *International Journal of Human-Computer Studies*, 40:1009-1032, 1994.
- [11] D.L. Parnas. Using mathematical models in the inspection of critical software. In M. Hinchey and J. Bowen, editors, *Applications of Formal Methods*. Prentice Hall, 1995. ISBN 0-13-366949-1.
- [12] D. Robertson and J. Agusti. *Software Blueprints: Lightweight Uses of Logic in Conceptual Modelling*. Addison Wesley, 1999. in press.
- [13] D. Robertson, A. Bundy, R. Muetzelfeldt, M. Haggith, and M. Uschold. *Eco-Logic: Logic-Based Approaches to Ecological Modelling*. MIT Press (Logic Programming Series), 1991. ISBN 0-262-18143-6.
- [14] M. Uschold and M. Gruninger. Ontologies: Principles, Methods and Applications. *Knowledge Engineering Review*, 11(2):93-136, 1996. ISSN 0269-8889.

Automated Verification of Function Block Based Industrial Control Systems

Norbert Völker
Department of Computer Science
University of Essex
Colchester CO4 3SQ, UK
+44 1206 872258
nvoelker@essex.ac.uk

Bernd J. Krämer
FernUniversität
Faculty of Electrical Engineering
D-58084 Hagen, Germany
+49 2331 987-371
bernd.kraemer@fernuni-hagen.de

Abstract

IEC 61131-3, the world-wide standard for industrial control programming, is increasingly being used in safety-related applications. They include safety-instrumented functions, such as burner management, emergency shutdown and gas detection, but also complex automation processes controlling, e.g., a chemical production plant. Testing techniques, however, which predominate in today's practice, cannot demonstrate the functional correctness and safety of an application under all operating conditions, in general.

In this paper we present a theorem prover-based technique supporting software verification during design as supplementary validation measures. It applies to individual function blocks and to networks of function blocks. The former provide re-usability from macro to micro level and are usually maintained in domain-specific component libraries, the latter represent complete control loops. Based on this distinction, the correctness and safety verification task can be separated into a single, a priori verification of each library component and a separate compositional proof of individual application programs. We briefly describe the semantic embedding of three most used languages of the IEC standard in a suitable logic and present our verification approach. We conclude with a sketch of design ideas for a verification tool usable by engineers in industry and safety licensing authorities.

Keywords

Safety-critical control systems, dependable software, PLC programming, IEC 61131-3, modular verification, higher order logic theorem proving.

1 INTRODUCTION

Programmable logic controllers (PLCs) are forming a growing market of special purpose hybrid systems integrating micro-electronic and software components. PLCs are particularly suited to solve application problems in machine logic, process automation, manufacturing, and data acquisition. They were developed to replace traditional hard-wired switching networks based on relay or discrete electronic logic.

The rapid development of PLC systems in the eighties led to a wealth of incompatible vendor-specific PLC programming languages within the process industries impeding the design of more complex, open and distributed control applications. In response to this situation, the international standard IEC 61131-3 for PLC programming [14] was developed and is currently¹ under review by the IEC [5]. The standard applies to a wide range of programmable controllers and harmonizes the way engineers look at industrial control by standardizing the programming interface.

The standard provides a class of five purpose-built languages that overlap conceptually and share a subset of programming elements [18]. Three languages of the standard, Function Block Diagram (FBD), Ladder Diagram (LD) and Sequential Function Chart (SFC) have a graphical appearance. FBD supports component-based application programming, while SFC is mainly used for depicting sequential behavior of a control system including alternative and parallel execution sequences.

New capabilities of PLCs, the comfort of the PLC languages, and strong economical demands led to the current situation that we are increasingly depending on PLC-based systems for control and automation functions in safety-related applications. Examples include (air) traffic control, patient monitoring, process automation, e.g. in chemical industry, emergency shut down systems in power generation, and production line control.

The growing awareness of our society of the need to protect the environment, a higher sensitivity to accidents caused by ill-designed technology or processes, and a declining trust in marketing statements of manufacturers produce an enormous pressure

¹1998

to increase the dependability of safety related applications. In practice, however, we observe that rigorous proof techniques and robust tools that can be used effectively by practitioners in industry and regulatory authorities and in the application domain are not available. Existing design guidelines and testing practices may help to detect design and programming errors but they cannot guarantee the absence of faults that may cause a disaster because exhaustive testing is limited to rare cases.

In the main body of this paper we explore function blocks – which represent the engineer's idea of re-usable "software ICs" – and sequential function charts to develop a modular, theorem prover-based verification framework. By taking components from application-specific libraries of verified standard function blocks, the verification of new applications is reduced considerably because only the correctness of the composition has to be established for each new application.

In the following section we introduce core concepts of FBD and SFC. In Section 3 we argue about the logic and theorem proving assistant used to verify functional correctness and safety of individual function blocks and complete control applications built from such components. This verification process can be automated by a semantic embedding of the selected PLC languages into that logic. This embedding is explained in Section 4, while our verification approach and the challenges of handling complex continuous systems are sketched in Section 5 and Section 6. We conclude with a brief summary and an outlook on an industrial strength verification tool that is to build on this or a similar verification framework and can ultimately be used by domain experts with little or no expertise in software verification.

2 FUNCTION BLOCKS AND SEQUENTIAL FUNCTION CHARTS

Function blocks are program organization units with a private state that persists from one invocation to the next. A function block interacts with its environment primarily via input and output variables. The standard also allows global variables but our verification framework does not support these. Besides keeping the semantics simple, this also has the advantage that the execution of function blocks has no side-effects.

From a semantic point of view, function blocks are a special case of deterministic reactive modules [1]. According to the model of reactive systems [9], their execution takes place in a sequence of rounds. At the start of each round, the input variables are read. This is followed by an update of the private and output variables. This update is functionally dependent on the current value of the input variables and the previous state of the private and output variables.

The description of a function block can be split into the declaration of its external interface and a specification of the internal implementation. The former is part of the function block signature that specifies the types and names of variables including local instances of function blocks. In the context of graphical representations, the input and output variables will also be referred to as ports. The interface specification is similar to the description of interfaces in other languages such as CORBA-IDL [8]. The internal implementation of a function block body can be carried out in any of the five IEC 61131-3 programming languages or even in other languages such as C or Java.

As an example, Figure 1(a) shows a graphical representation of the external interface of the function block DEBOUNCE taken from the IEC 61131-3. DEBOUNCE has two input variables IN and DB_TIME of type BOOL and TIME and two output variables OUT and ET_OFF of the same types.

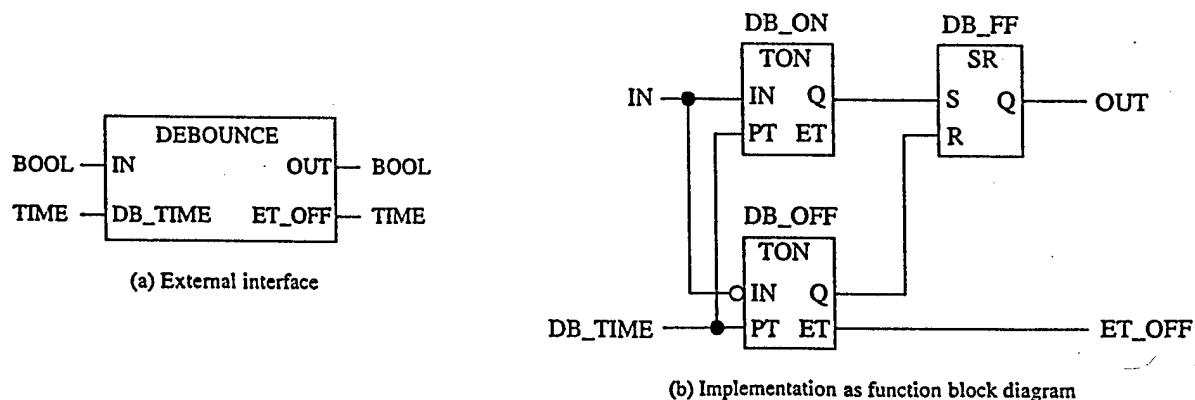


Figure 1: Function block DEBOUNCE

An implementation of DEBOUNCE as a function block diagram is depicted in Figure 1(b). The function blocks DB_ON and DB_OFF are two separate instances of the timer function block TON, while DB_FF is an instance of the SR flip-flop included

in the standard. By connecting input and output ports, a diagram is "wired together" from the components. As in the graphical representation of circuits, at the open circle such as at the input port IN of DB_OFF indicates the negation of a Boolean signal. The named instances of function blocks will usually also be referred to simply as function blocks. The function block DEBOUNCE is composed from standard function blocks predefined in the norm. Such a composite function block can itself be used in further applications just as if it was one of the standard function blocks. This feature is useful for building an in-house or domain specific collection of function blocks.

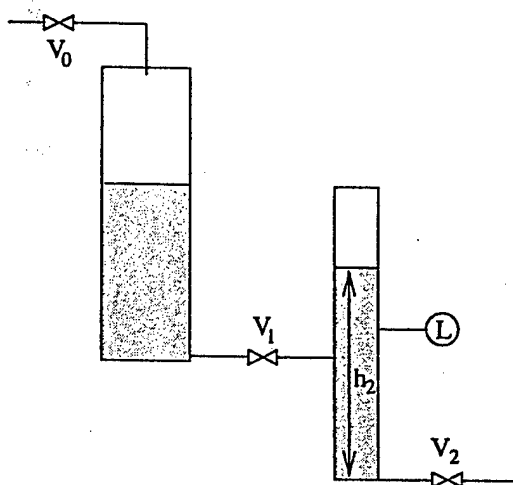
The textual IEC 61131-3 language ST is similar in appearance to a structured programming language such as PASCAL. Figure 2 shows an alternative implementation of the body of DEBOUNCE in ST.

```
DB_ON (IN := IN, PT := DB.TIME);
DB_OFF (IN := NOT IN, PT := DB.TIME);
DB_FF (S := DB_ON.Q, R := DB_OFF.Q);
OUT := DB_FF.Q;
ET_OFF := DB_OFF.ET;
```

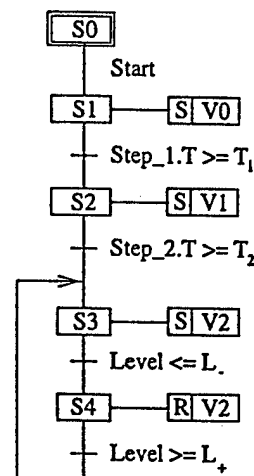
Figure 2: DEBOUNCE in Structured Text

The second graphical language of the standard, SFC, can be regarded as an application of Petri nets. Its language concepts include transitions, steps and actions. They serve to co-ordinate the execution of function blocks that are regarded as asynchronous sequential processes.

To illustrate the role of SFC, we refer to a small laboratory plant that has been used previously as a benchmark for non-linear control design methods [13] and for the tool-aided analysis of discretely controlled continuous systems [15]. The plant features two cylindric tanks that are located at different levels. The tanks are equipped with three pipes and three valves controlling the flow of liquid between the tanks, at the inlet and at the outlet (see Fig. 3(a)). The pipes are controlled by valves V_0 , V_1 and V_2 . The liquid level in the second tank is measured by a sensor L . A core safety requirement for this application is to avoid overflow in the coupled tank system.



(a) Laboratory plant



(b) SFC controller

Figure 3: Laboratory plant with SFC controller

The SCF depicted in Fig 3(b) controls the behavior of the system. It consists of five steps $s_0, \dots, s_4$. The actions connected with the steps control the state of the valves: the qualifiers S and R denote setting and resetting of an action, respectively. The transitions separating the steps are enabled by Boolean valued expressions that reflect conditions on the state of the associated function block.

The encapsulation provided by function blocks together with their openness with respect to the internal implementation furthers the reuse of function blocks in different applications. Hence, it makes sense to develop component libraries. Examples include the collection of standard function blocks of the IEC 61131-3, the German standard norm [30], and the domain specific library of function blocks used by a German manufacturer of chemicals and drugs we have studied earlier. This in-house library consists of about 70 function blocks that are sufficient to specify/program chemical process automation tasks.

3 HIGHER ORDER LOGIC FOR VERIFICATION

The basic logic underlying our verification approach is higher order logic [3, 7]. There are several good reasons for this choice:

1. The means of abstraction and quantification over functions make this logic very expressive and thus well suited to the concise description of complex theories. Evidence of this fact is provided by the embedding of hardware description languages [4] and the verification of floating point algorithms [12].
2. HOL is a widely studied and well understood logical system with a remarkably small number of axioms and inference rules. Its expressiveness makes it possible to use definitional extension as the principal method of theory development. Since this method is conservative, logical inconsistencies can be practically ruled out.
3. Automatic type inference systems for HOL make type annotations to a great extent unnecessary. This shortens formulas and proofs because the information contained in the typing is automatically inferred and propagated.

In comparison to alternatives such as Zermelo-Fränkel set theory, there are also a few disadvantages:

1. The type discipline of HOL leads to a certain loss of flexibility, cf. [17]. This statement remains true despite the expressiveness of polymorphism and symbol overloading available in systems such as Isabelle/HOL.
2. In comparison with first and second order logics, the implementation of the HOL type system is technically more demanding. In particular, the existence of type and function variables complicates unification, the basic method of equation solving [22]. Also, most research in automated theorem proving has been performed in the area of first order theories.

For our purpose, the advantages of HOL outweigh these drawbacks. Its extendibility makes it unnecessary to introduce special logics for the definition of the semantics of programs and specifications. Instead, HOL provides a logical core that can serve as the common semantical basis for a range of different formalisms.

Furthermore, it is important that the logic is supported by several reliable and efficient mechanical theorem proving assistants. Our system of choice is the object logic HOL of the generic theorem proving assistant Isabelle [25]. Like the HOL system [7], Isabelle builds on the functional programming language SML [20]. Noteworthy alternatives include the HOL system and the LISP based PVS [27] system.

With regard to verification, the high degree of safety and reliability of a proof assistant are of paramount importance. In the Isabelle system, a number of measures are taken in order to achieve this aim:

1. Theorems are elements of a special abstract SML data type `thm`. New elements of this type can only be formed by a small number of operations representing valid logical deductions or explicit axioms. If one assumes that the Isabelle implementation of these basic operations is correct, then the static type checking of SML guarantees also the logical validity of all derivations.
2. The preferred method for extending theories is definitional. This minimizes the danger of logical inconsistencies.
3. Isabelle is an open and extendible system with a freely available source code. The source code is well structured and written in a functional programming language with only little use of imperative features. This makes it open to be scrutinized by independent researchers.

Furthermore, Isabelle has a comprehensive international user community. These combined factors have given Isabelle - like the HOL system - the reputation of an extremely trustworthy proof support system.

In addition to safety, a high degree of proof automation is essential in order to cope in a reasonable time with the many proof obligations arising during verification. The main tools of the Isabelle systems in this respect are a term-rewriting simplifier and a proof search tool called the classical reasoner. External decision procedures can be invoked from Isabelle using an oracle mechanism. The degree of automation is sufficient for the definition of formal semantics and the verification of small to medium-sized function block applications.

4 THE EMBEDDING OF FUNCTION BLOCKS IN HOL

The main motivation behind the formulation of higher order logic as used in the HOL system was the mechanical verification of hardware. Remarkable achievements in this area include the verification of an ATM network component [6] and of RISC pipeline conflicts [29]. In comparison, success of HOL in the area of software verification has been more tedious. Research has concentrated up-to now on particular aspects of real programming languages such as the type safety of a Java subset [23].

The foundation of our verification framework is a HOL embedding of a subset of Structured Text (ST). The technical details of this embedding can be found in [31]. It is a relatively deep embedding, which means that the syntax of function blocks and the assignment of semantics are represented explicitly in HOL. Semantics are defined via evaluation functions for the four different syntactical categories, namely expressions, statements, functions and function blocks. As a result, every function block is associated with a deterministic, but not necessarily finite Mealy automaton in HOL. Time is treated as an input variable. Like all other input variables, its value stays constant in each round. This fits well with the paradigm of reactive systems which produce responses instantaneously.

The HOL terms that describe the semantics of function blocks are initially cluttered with occurrences of the evaluation functions. In a term rewriting process which resembles a symbolic evaluation, these occurrences can be eliminated. This process can be largely automated. It yields HOL terms that resemble simulations of ST function blocks viewed as functional programs. In this form, the automata are suitable for verification.

An important aspects of our semantics is compositionality. This means that the transition function of the automaton belonging to a composed block is a composition of the transition functions of the automata belonging to the components. Thus proven properties of the components can be reused. Furthermore, by abstracting over component properties, it is possible to prove properties of composed function blocks without reference to the concrete implementation of the components.

In addition to ST, our verification framework also deals with subsets of the two graphical IEC 61131-3 languages SFC and FBD. This is based on interpretations of these two formalisms in ST. The result is in both cases a formal semantics that is sequential and deterministic. We will sketch the interpretation of function block diagrams below. For SFC, we refer to [31].

Interpretation of Function Block Diagrams in ST

The connection of function block inputs with outputs in a diagram induces a dependency relation on its components: a function block *A* depends on a function block *B* provided at least one input port of *A* is connected directly or indirectly to some output port of *B*. The relation is a partial order as long as the diagram does not contain feedback loops such as in Fig. 4(a). In the latter case, we require the user to specify feedback variables for connections. This has the effect of a unit delay on the connections involved, i.e., the input port always receives the output value from the previous round. In the definition of the dependency relation for function block diagrams with feedback variables, such delayed connections are disregarded. This eliminates cycles and ensures that the dependency relation is a partial ordering.

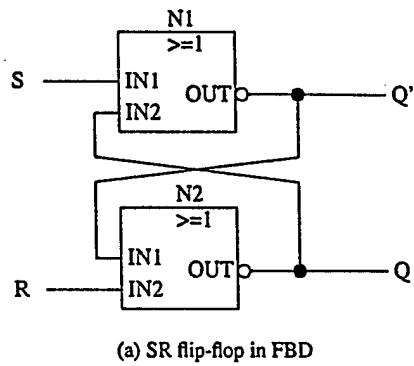
The essential step in the interpretation of a function block diagram in ST is a sequentialization of the function block executions per round. The only requirement placed on this sequentialization is its compatibility with the dependency ordering. This rule does not specify the relative execution order of independent function blocks. For example, in the function block DEBOUNCE, both DB_ON and DB_OFF have to be executed before DB_FF, but nothing is said about their relative execution order. Since we disallow global variables in our verification framework, this under-specification does not lead to non-determinism; in addition, the resulting semantics of a function block is not affected by the choice of execution sequence as long as it is compatible with the dependency ordering.

A chosen execution order can be translated directly to a sequence of ST function block invocations. This is followed by updates of feedback and output variables. Figure 4(b) exemplifies this for the case of the SR flip-flop. Here, feedback variables FB1 and FB2 have been introduced for the connections from N1.OUT to N2.IN1 and N2.OUT to N1.IN2.

5 THE VERIFICATION APPROACH

The deep embedding of PLC programming languages in HOL provides a formal semantics. Furthermore, the semantics given above are operational. Function blocks can thus be evaluated symbolically using a term rewriting tool. Requirements on the behavior of function blocks can be translated to HOL predicates and proven formally. Figure 5 shows the verification process for SFC function blocks using linear time temporal logic (LTL, [19]) as specification language.

One of the strong points of the HOL based approach is its openness with respect to possible extensions. An addition of further programming or specification language constructs is unproblematic as long as it does not affect the underlying semantical model of the already embedded language parts. The same remark holds for the modeling of machine or environment aspects, which might be necessary for the verification of more complex systems. To put it more generally: HOL serves as logical glue that connects different programming and specification formalism and allows their integration and analysis within one framework.



```

N1 (IN1 := S, IN2 := FB2);
N2 (IN1 := FB1, IN2 := R);
FB1 := N1.OUT;
FB2 := N2.OUT;
Q'  := FB1;
Q   := FB2

```

(b) SR flip-flop in ST

Figure 4: Function block with feedback loop

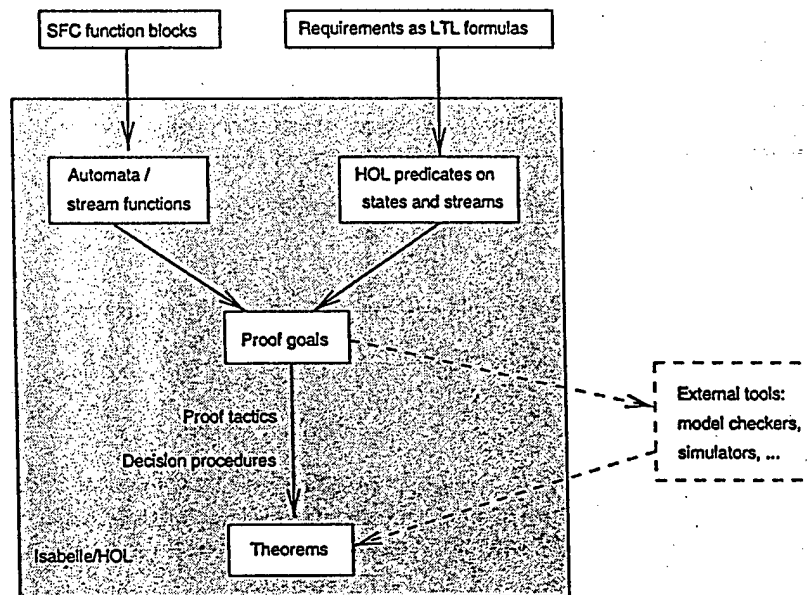


Figure 5: Function Block Verification Process

It might be interesting to investigate the integration of the Step system developed by Zohar Manna's group in Stanford with the framework used here. Especially their combination of model checking and theorem proving might be advantageous for the application domain considered in this contribution.

In relatively small examples such as the verification of a liquid container controller presented in [16], the standard Isabelle/HOL proof tools are sufficient. Because specifications are mapped to predicates on streams, the basic proof principle is induction over the natural numbers. In the induction step, the validity of a statement in round $(n + 1)$ has to be derived from its validity in round n . Induction is also essential for the proof of auxiliary algebraic equalities and inequalities and the verification of iterated structures such as a generic adder. Other frequent proof techniques are case distinctions, algebraic simplifications and arithmetic estimations. Isabelle's classical reasoner has been very useful for the automation of these kinds of proofs.

For more complex applications, a higher degree of proof automation is essential. This starts off with the automated translation of function blocks into Isabelle theories. Tactics specially adapted to programming or specification language constructs should be tried automatically or offered interactively to the user for selection and parameterization. Relevant automated proof procedures include the symbolic model checking of finite systems [2] and algorithms for establishing program invariants [28, 10].

6 THE CHALLENGE OF COMPLEX DYNAMICS

Up-to now, the use of theorem prover based tools has been restricted to the verification of systems with relatively simple continuous dynamics. This is partly due to the fact that the treatment of such systems would require extensive real/complex analysis libraries. As the pioneering work in [11] shows, this is a comprehensive task. Even with such libraries, a complete analytic verification of systems such as the two tank laboratory plant sketched in Fig. 3 seems a daunting task.

Besides providing formal models of controllers and abstractions of plant properties, one useful role for deductive proof tools in this area might be the validation of interpolation and extrapolation properties. These guarantee that nothing unexpected happens for parameter combinations that have not been explicitly covered during simulation or model checking. This validates intuitive worst-case reasoning and increases the trustworthiness of verification results.

7 TOWARDS AN INDUSTRIAL STRENGTH TOOL

In the main body of this paper we presented a theorem prover based flexible verification technique that supports modular proofs of PLC programs expressed in FBD, SFC and ST. In general, the development of such proofs with the help of theorem prover assistants requires high skills from the quality assurance personnel because the proof assistant relies on sophisticated user guidance. These skills cannot be expected from engineers in the field.

Conversely, people with skills in formal specification and verification techniques normally lack the domain expertise needed to understand functional and safety requirements that are often not made explicit and, if so, are usually presented in an incomplete, ambiguous and informal manner. In the course of our work with the IEC standard, its German counterpart, and the in-house standard and function block library of a manufacturer in chemical industry we spent days and weeks in reading through these documents and many hours talking to domain experts to fully understand the requirements.

Hence, to make the verification approach we presented work in automation practice, we need to find effective means to solve the following three tasks:

1. Comprehensive elicitation of functional, safety, and – if appropriate – timing requirements.
2. Formalization of these requirements in a suitable logic.
3. Correctness proof.

As the set of standard function blocks that are typically used in specific control domains ranges between 50 and a few hundreds and the complexity of the majority of function blocks maintained in the domain library is relatively low, the effort to have these tasks performed by computer theoreticians is acceptable. It needs to be summoned up only once.

However, for handling individual control applications that are composed of networks of function blocks, we need to wrap an open verification environment with a front-end that is usable by domain experts. This verification environment may build on a theorem prover as its backbone and comprise other tools such as model checkers, simulators or computer algebra systems. The interface to the front-end must be capable to elicit enough facts about critical application requirements through a series of communication interactions with domain experts such that formal requirement statements can be derived. Such dialogs need to know about the terminology of the field, they may rely on a collection of known requirements typical for that domain, and they may exploit proven properties of function blocks connected to the application interface and the inner wiring of the application program to conduct that dialog. It may also exploit paraphrasing capabilities to verify the adequacy of formalized requirement

statements acquired in earlier communications. The work on knowledge intensive software engineering tools conducted by Rich and Waters (cf., e.g. [26]) might provide prototype solutions for the engineering environment sketched here. In a related project in South Africa, we have also used Parnas' table specification technique [24] to formalize standard function block interfaces [21] in a way that might be more readable to engineers.

To facilitate the verification task, it is very important to find proof patterns and reusable proof strategies to automate recurring verification steps. In this respect, the integration of automatic model checking procedures such as pioneered by N. Shankar for the PVS system seems particularly promising.

To come up with usable solutions, a close co-operation with interested vendors, users and evaluators for PLC controllers in safety critical fields is urgently needed.

REFERENCES

- [1] R. Alur and T. Henzinger. Reactive modules. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 27–30 1996.
- [2] R. Alur, T. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.
- [3] P. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Academic Press, 1986.
- [4] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. Melham, and R. Boute, editors, *Theorem Provers in Circuit Design. Proceedings of the IFIP TC10/WG 10.2 International Conference, Nijmegen, June 1992*. North-Holland, 1992.
- [5] I. E. Commission. On-line at <http://www.iec.ch>.
- [6] P. Curzon. The Formal Verification of the Fairisle ATM Switching Element. Technical Report 329, Computer Laboratory, University of Cambridge, 1994.
- [7] M. Gordon and T. Melham. *Introduction to HOL*. Cambridge University Press, 1993.
- [8] O. M. Group. CORBA/IIOP 2.2 Specification. Online at <http://www.omg.org>, 1998.
- [9] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.
- [10] N. Halbwachs, Y. Proy, and P. Roumanoff. Verification of Real-Time Systems using Linear Relation Analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.
- [11] J. Harrison. *Theorem Proving with the Real Numbers*. PhD thesis, Computer Laboratory, University of Cambridge, 1996.
- [12] J. Harrison. Floating point verification in HOL Light: the exponential function. Technical Report 428, Computer Laboratory, University of Cambridge, 1997.
- [13] T. Heckenthaler and S. Engel. Approximately time-optimal control of a two-tank system. *IEEE Control Systems Magazine*, 14(3), 1994.
- [14] IEC International Standard 1131-3. *Programmable Controllers. Part 3: Programming Languages*, 1993.
- [15] S. Kowalewski, M. Fritz, H. Graf, S. S. J. Preußig, O. Stursberg, and H. Treseler. A case study in tool-aided analysis of discretely controlled continuous systems: the two tanks problem. In *Presented at the 5th Int. Workshop on Hybrid Systems (HS V), Notre Dame, USA, September 1997*.
- [16] B. Krämer and N. Völker. A highly dependable computer architecture for safety-critical control applications. *Real-Time Systems Journal*, 13:237–251, 1997.
- [17] L. Lamport and L. Paulson. Should your specification language be typed? Technical Report 425, Computer Laboratory, University of Cambridge, May 1997.
- [18] R.W. Lewis. *Programming industrial control systems using IEC 1131-3*. The Institution of Electrical Engineers, London, 1995.
- [19] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

- [20] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [21] S. Morton. Specifying VDI/VDE 3696 Function Blocks. Master Thesis, University of Witwatersrand, Johannesburg 1998.
- [22] T. Nipkow. Higher-order unification, polymorphism, and subsorts. In *Proc. 2nd Int. Workshop Conditional and Typed Rewriting Systems*. LNCS 516, 1991.
- [23] T. Nipkow and D. von Oheimb. Java_{light} is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, 1998.
- [24] D. Parnas, J. Madey, and M. Iglewski. Formal Documentation of Well-Structured Programs. Technical Report CLR 259, Faculty of Engineering, McMaster University, Hamilton, Ontario 1992
- [25] L. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.
- [26] C. Rich and R.C. Waters. Knowledge Intensive Software Engineering Tools. *IEEE Transactions on Software Engineering*, 4(5):424–430, 1992
- [27] J. Rushby and D. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report SRI-CSL-95-10, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995. Revised, July 1996. Available, with specification files, from <http://www.csl.sri.com/csl-95-10.html>.
- [28] H. Saïdi. The Invariant Checker: Automated deductive verification of reactive systems. In O. Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
- [29] S. Tahar and R. Kumar. Formal Specification and Verification Techniques for RISC Pipeline Conflicts. *The Computer Journal*, 38(2):111–120, 1995.
- [30] VDI/VDE Richtlinie 3696 *Herstellerneutrale Konfigurierung von Prozeßleitsystemen; Allgemeines zur herstellernneutralen Konfigurierung*, 1995.
- [31] N. Völker. *Ein Rahmen zur Verifikation von SPS-Funktionsbausteinen in HOL*. PhD thesis, FernUniversität Hagen, Shaker Verlag, 1998.

The Story of Re-engineering of 350,000 Lines of FORTRAN Code *

M. Shing, Luqi, V. Berzins, M. Saluto, J. Williams, J. Guo, and B. Shultes

Computer Science Department

Naval Postgraduate School

{mantak, luqi, berzins, saluto, williamj, gj, shultesb}@cs.nps.navy.mil

Abstract

This paper describes a case study to determine whether computer-aided prototyping techniques provide a cost-effective means for re-engineering legacy software. The case study consists of developing a high-level modular architecture for the existing US Army Janus combat simulation system, and validating the architecture via an executable prototype using the Computer Aided Prototyping System (CAPS), a research tool developed at the Naval Postgraduate School. The case study showed that prototyping can be a valuable aid in re-engineering of legacy systems, particularly in cases where radical changes to system conceptualization and software structure are needed [1]. The CAPS system enabled us to do this with a minimal amount of coding effort.

1. Introduction

Re-engineering is typically needed when a system performing a valuable service must change, and its current implementation can no longer support cost-effective changes. Such legacy systems usually lack accurate documentation, modular structure, and coherent abstractions that correspond to current or projected requirements. Past optimizations and design changes have spread design decisions that must be changed over large areas of the code. The main objective of a re-engineering effort is thus to develop a coherent modular architecture that can support cost-effective change and to transform the legacy implementation to fit into the new architecture. The Janus system fits this classical situation.

Janus(A) is a software-based war game that simulates ground battles between up to six adversaries. It is an interactive, closed, stochastic, ground combat simulation with color graphics. Janus is "interactive" in that command and control functions are entered by military

analysts who decide what to do in crucial situations during simulated combat. The current version of Janus operates on a Hewlett Packard workstation and consists of a large number of FORTRAN modules, organized as a flat structure and interconnected with one another via FORTRAN COMMON blocks, resulting in a software structure that makes modification to Janus very costly and error-prone. There is a need to modernize the Janus software into a maintainable and evolvable system and to take advantage of modern Personal Computers to make Janus more accessible to the Army. The Software Engineering group at the Naval Postgraduate School was tasked to extract the existing functionality through reverse engineering and to create an object-oriented architecture that supports existing and required enhancements to Janus functionality.

2. The Re-engineering Process

2.1. Reverse-Engineering

The first step in reverse-engineering is system understanding. Analysis of the legacy implementation is a daunting but inescapable part of this step. If printed out at 60 lines per page, 350,000 lines would fill almost 6000 pages. We recoiled from the magnitude of this effort, but in hindsight, this was a mistake that slipped the schedule of the project by several months. Understanding a design of this complexity requires time for mental digestion, even with tool support and judicious sampling. We should have started analysis of the code right away and should have persistently continued this task in parallel with all other re-engineering activities. Cross fertilization between all the tasks would have helped us recognize some dead-end directions earlier and would have enabled us to spend meeting time more effectively. However, we actually started the process with a series of brief meetings with the client, TRAC-Monterey, asking questions and making notes on the system's operation and its current functionality. We paid attention to the client's view of the system to gather their ideas on its strengths, weaknesses, and desired and undesired functionality.

* This research was supported in part by the U.S. Army Research Office under grant number 35037-MA and in part by a grant from the U. S. Army Training and Doctrine Analysis Command.

These meetings were indispensable because they gave us information that was not present in the code. Additionally we collected copies of the Janus User's manual, the Janus Programmer's Manual, the Janus Database Management Program Manual, the Janus Software Design Manual, and the Janus Algorithm Document [2, 5-8]. These documents helped us get started because they contained higher level information and were much shorter than the code. They were also older, and it was a constant struggle to determine which parts were still accurate, and which were not.

Since we were not familiar with the domain of ground combat simulation, we were using these meetings to determine the requirements of this domain, often playing the role of "smart ignoramuses" [10]. Domain analysis has been identified as an effective technique for software re-engineering [11]. Our experience suggests that competent engineers unfamiliar with the application domain have an essential role in re-engineering as well as in requirements elicitation because lack of inessential information about the application domain makes it easier to find new, simpler design structures and architectural concepts to guide the re-engineering effort.

The next step is to abstract the system's functionality and produce system models that would most accurately represent that functionality [1]. Armed with the Janus source code, we proceeded to divide the code by directories amongst the team members. Each team member was assigned roughly six to seven directories to explore, examine and gather information. Using manual techniques augmented with UNIX commands and review procedures, we were able to get a fairly good idea of what each subroutine was designed to do. We also used the Software Programmers' Manual [5] to aid in understanding each subroutine's function. In doing so we were able to group the subroutines by functionality to get a better understanding of the major data flows between programs. Using that knowledge, we developed functional models from the data flows.

We used the Computer-Aided Prototyping System (CAPS) [3,4], an automated tool developed at the Naval Postgraduate School, to assist in developing the abstract models. CAPS allowed us to rapidly graph the gathered data and transform it into a more readable and usable format. Additionally, CAPS enabled us to concurrently develop our diagrams, and then join them together under the CAPS environment, where they can be used to generate an executable model.

Figure 1 shows the resultant top-level structure of the existing Janus system. It consists of five subsystems - *cs_data_mgmt*, *scenario_db*, *janus*, *jaaws*, and *postp*. The *cs_data_mgmt* subsystem manages combat system databases. The *scenario_db* subsystem manages the different scenarios and simulation runs in the system. The

janus subsystem simulates the ground battles. The *jaaws* subsystem allows analysts to perform post-simulation analysis and the *postp* subsystem allows Janus users to view simulation reports.

2.2. Transformation of Functional Models to Object Models

Next, we developed object models of the Janus System using the aforementioned materials and products, to create the modules and associations amongst them. Information modeling is needed to support effective re-engineering of complex systems [12]. This was probably the most difficult and most important step. It required a great deal of analysis and focus to transform the currently scattered sets of data and functions into small, coherent and realizable objects, each with its own attributes and operations. In performing this step, we used our knowledge of object-oriented analysis and applied the OMT techniques and the UML notations to create the classes and associated attributes and operations. This was a crucial step because we had to ensure that the classes we created accurately represented the functions and procedures currently in the software. Restructuring software to identify data abstractions is a difficult part of the process. Transformations for meaning-preserving restructuring can be useful if tool support is available [13]. We used the HP-UNIX systems at the TRAC-Monterey facility to run the Janus simulation software to aid in verifying and/or supplementing the information we obtained from reviewing the source code and documentation. This step enabled us to better analyze the simulation system, gaining insight into its functionality and further concentrate on module definition and refinement.

2.3. Refinement and Validation of the Object Models

During this phase of the project, the re-engineering team met several times each week for a period of two and a half months to discuss the object models for the Janus core elements and the object-oriented architecture for the Janus System. They presented the findings to the Janus domain experts at least once per week to get feedback on the models and architectures being constructed. In addition, the re-engineering team also presented the findings to members of the OneSAF project, the Combat21 project, and the National Simulation Center. We found that information from these domain experts was essential for understanding the system, particularly in cases where the legacy code did not correspond to stakeholder needs. This supports the hypothesis advanced

in [14] that the involvement of domain experts is critical for nontrivial re-engineering tasks. Based on the feedback from the domain experts, the re-engineering team revised the object models for the Janus core elements and developed a 3-tier object-oriented architecture for the Janus System (Figure 2).

3. Software Architecture for the Janus Combat Simulation Subsystem

Central to the existing Janus Combat Simulation Subsystem is the program RUNJAN, which is the main event scheduler for the Janus simulation. RUNJAN determines the next scheduled event and executes that event. If the next scheduled event is a simulation event, RUNJAN will advanced the game clock to the scheduled time of the event and perform that event. The existing event scheduler uses global arrays and matrices to maintain the attributes of the objects in the simulation. Hence, one of the major tasks in designing an object-oriented architecture for the Janus combat simulation subsystem was to distribute the event handling functions to individual objects. Moreover, it was necessary to redefine some event categories in order to provide a uniform framework to eliminate redundant coding of the same or similar functions and to take advantage of dynamic dispatching of event handling functions in the object-oriented architecture.

Interactions between the simulation engine and the world modeler (the distributed simulation network) are performed implicitly within the various event handlers in the existing Janus. Such interactions are made explicit in the new architecture in order to provide a uniform framework to update World Model objects during the simulation.

The new architecture uses an explicit priority queue of event objects to schedule the simulation events. Each event object has an associated simulation object, which is the target of the event. There are 14 event groups, which correspond to the 14 event subclasses shown in Figure 3.

An object-oriented approach enabled us to reduce the number of event types needed in the simulation. Depending on the subclass that an event object belongs to, the "execute" method will invoke the corresponding event handler of the associated simulation object to handle the event (Figure 4). The simulation object superclass defines the interface of the event handlers for the event groups, and provides an empty body as the default implementation for the event handlers. The methods are overridden by the actual event handler code at the subclasses that have non-empty actions associated with the events.

The above architecture enables a very simple

realization of the main simulation loop:

```

initialization;
While not_empty(event_queue) loop
    e := remove_event(event_queue);
    e.execute();
End loop;
finalization;

```

Note that this same code handles all kinds of events, including those for future extensions that are yet to be designed. Event objects are created and inserted into the event queue by the initialization procedure at the beginning of the simulation, by the constructors of new simulation objects, and by the actions of other event handlers. Depending on implementation decisions regarding when and how events are inserted into the priority event queue, it may be necessary to allow events to change their priorities while waiting in the queue.

World Model object subclasses were created to provide specialized methods for the world modeler to update the objects from other simulators. Information concerning objects local to the Janus simulator can be broadcast over the simulation network either periodically by an active world modeler object, or by individual local objects whenever they update their own states.

4. Development of an Executable Prototype Using CAPS

In order to validate the proposed architecture and to refine the interfaces of the Janus subsystems, we developed an executable prototype using CAPS. Figure 5 shows the top-level structure of the prototype, which has four subsystems: Janus, GUI, JAAWS and the POST-PROCESSOR. Among these four subsystems, the Janus and the GUI subsystems (depicted as double circles) are made up of sub-modules shown in Figures 6 and 7, while the JAAWS and the POST-PROCESSOR subsystems (depicted as single circles) are mapped directly to objects in the target language. After we entered the prototype design into CAPS, we used the CAPS execution support system to generate the code that interconnects and controls these subsystems.

Due to time and resource limitations, we only developed the prototype for a very small simulation run, which consists of a single object (a tank) moving on a two-dimensional plane, three event subclasses (move, do_plan, and end_simulation), and one kind of post-processing statistics (fuel consumption). In addition, a simple user interface was developed using CAPS/TAE [9] (Figure 8). The resultant prototype has over 6000 lines of program source code, most of which was automatically generated, and contains enough features to exercise all parts of the architecture. The code that handles the motion

of a generic simulation object was very simple, but it was designed so that it would work in both two and three dimensions without modification (currently the initialization and the movement plan of the tank object never call for any vertical motion). The code was also designed to be polymorphic, just as was the main event loop. This means the same code will handle the motion of all kinds of simulation objects without any modifications, including new types of simulation objects that are part of future enhancements to Janus and have not yet been designed or implemented.

5. Lessons Learned

Our prototyping experiment showed that the proposed object-oriented architecture allows design issues to be localized and provides easy means for future extensions. We started out with a prototype consisting of only two events subclasses (move and end_simulation) and were able to add a third event subclass (do_plan) to the prototype without modifying the event control loop of the Janus combat simulator.

We also demonstrated the use of inheritance and polymorphism to efficiently extend/specialize the behavior of combat units. For example, to implement the move_update_object method of a tank subclass which uses the general-purpose method from its superclass to compute its distance traveled and a specialized algorithm to compute its fuel consumption, we simply include 1 statement to invoke the move_update_object method of its superclass followed by three lines of code to update its fuel consumption. Moreover, other combat unit subclasses can be added easily to the prototype without the need to modify the event scheduling/dispatching code and usually without modifying existing event handlers.

The prototype also resulted in the following refinements to the proposed architecture:

- (1) Instead of a procedure with no return value, change the Execute_Event operation to return the time at which the next event is to be scheduled for the same simulation object, and introduce a special time value "NEVER" to indicate that no next event is needed. The proposed change turns the communication between the event dispatcher and the simulation objects from a peer-to-peer communication into a client-server communication. This change eliminates the need for the simulation objects to know the details of the event queue and allows the event dispatcher to use a single statement to schedule all recurring events for all event types.
- (2) Instead of recording the history of a simulation run in terms of sets of data files, model the simulation history as a sequence of events. The proposed change

provides a simple and uniform way to handle history records for all events, and allows the same modular architecture to be used for real-time simulations as well as post-simulation analysis. It also eliminates the need for the write-status event in the legacy software. This approach provides the greatest possible resolution for the event histories, which implies that any quantity that could have been calculated during the simulation can also be calculated by a post-simulation analysis of the event history, without any loss of accuracy. The only constraint imposed by this design refinement is that the simulation objects in the events must be copied before being included in the simulation history, to protect them from further changes of state as the simulation proceeds. This constraint is easy to meet in a full-scale implementation because the process of writing the contents of an event object to a history file will implicitly make the required copy.

The prototyping effort also exposed a design issue – should null events appear in the event queue? A null event is one that does not affect the state of the simulation, such as a move event for an object that is currently stationary. The prototype version adopted the position that such events should not be put in the event queue, since this corresponds to current scheduling policies in Janus, and appears at first glance to improve efficiency.

Our experience with the development of the prototype suggests that this decision complicates the logic and may not in fact improve efficiency. In particular, the process create_new_events (see Figure 6) could be eliminated if we allowed null events. This process scans all simulation objects once per simulation cycle to determine if any dormant objects have become active, and if so, schedules events to handle their new activity. The alternative is to have the constructor of each kind of simulation object schedule all of its initial events, and to have each event handler specify the time of next instance of the same event even if there is nothing for it to do currently. Handlers might still set the time of its next event to NEVER in the case of a catastrophic kill; however this is reasonable only if it is impossible to repair or restore the operation of the units that have suffered a catastrophic kill.

The reasons why this design change may improve efficiency in addition to simplifying the code are that:

- (1) the check for whether a dormant object has become active is done less often – once per activity of that object, rather than once per simulation cycle,
- (2) executing a null event is very fast – a few instructions at most, so the "unnecessary" null events will not have much impact on execution time, and
- (3) the computation to find and test all simulation objects

periodically would be eliminated.

One recommendation is to allow null events in the event queue, and to explicitly schedule every kind of event for every object unless it is known that there cannot be any non-empty events of that type in any possible state of the object. For example, under the proposed scheduling policy, immobile or irrecoverably damaged objects would not need to schedule future move events, but those that are currently at their planned positions would need to do so, because a change of plan would cause them to move again in the future, even though they are not currently moving.

6. Conclusion

Our experience in this case study suggests that prototyping can be a valuable aid in re-engineering of legacy systems, particularly in cases where radical changes to system conceptualization and software structure are needed.

In particular, we found that constructing even a very thin skeletal instance of the proposed new architecture raised many issues and enabled us to correct, complete, and optimize the architecture for both simplicity and performance.

The computer-aided prototyping tools in the CAPS system enabled us to do this with a minimal amount of coding effort. The bulk of the code was generated automatically, enabling us to concentrate on system structuring issues, to consider and evaluate various alternatives, and to improve the design while doing detailed manual implementation for only a few pages of critical code. Our experience corroborates the ideas that re-engineering is the combination of reverse engineering with forward engineering [15] and that the most useful representations for the information extracted via reverse engineering are those that can drive tools for forward engineering.

7. References

- [1] J. Williams, M. Saluto, *Re-engineering and Prototyping Legacy Software Systems-Janus Version 6.X*, MS Thesis, Naval Postgraduate School, March 1999.
- [2] *Janus Version 6 User's Manual*, Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.
- [3] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System", *IEEE Software*, 5(2), 1988, pp. 66-72.
- [4] Luqi, "System Engineering and Computer-Aided Prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1), 1996, pp. 15-17.
- [5] *Janus 3.X/UNIX Software Programmer's Manual*. Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas. Nov. 93.
- [6] *Janus 3.X/UNIX Software Design Manual*. Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas. Nov. 93.
- [7] *Janus Version 6 Data Base Manager's Manual*, Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.
- [8] J. Pimper and L. Dobbs, *Janus Algorithm Document*, Version 4.0, Lawrence Livermore National Laboratory, California, 1988.
- [9] B. Shultes and M. Shing, *A CAPS-TAE Tutorial*, Tech. Report NPSCS-99-008, Computer Science Department, Naval Postgraduate School, Monterey, CA, January 1999.
- [10] D. Berry, *Formal Methods: The Very Idea*, Some Thoughts About Why They Work When They Work*, Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems, 1999.
- [11] M. Moore and S. Rugaber, *Domain Analysis for Transformational Reuse*, Proceedings of 4th Workshop on Reverse Engineering, IEEE Computer Society, 1997.
- [12] O. Bray, M. Hess, *Re-engineering a Configuration-Management System*, IEEE Software, Vol. 12, No. 1, Jan. 1999.
- [13] V. Cabaniss, B. Nguyen, J. Moregenthaler, *Tool Support for planning the Restructuring of data Abstractions in Large Systems*, IEEE TSE, Vol. 24, NO. 7, July 1998.
- [14] S. Jarzabek, *Design of a Generic Reverse Engineering Assistant Tool*, Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95), 1995.

[15] I. Baxter, M. Mehlich, *Reverse Engineering is Reverse Forward Engineering*, Proceeding 4th Workshop on Reverse Engineering, IEEE Computer Society, 1997.

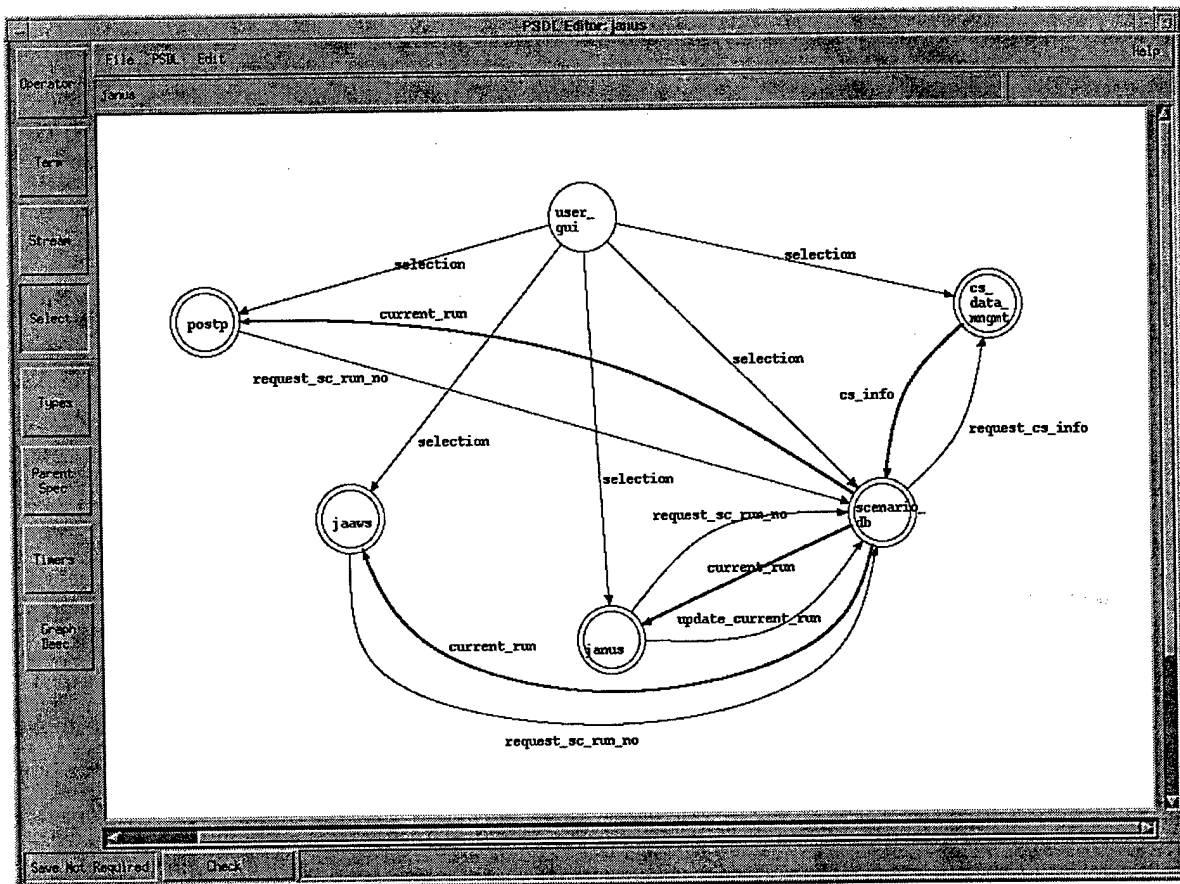


Figure 1. Top-level communication structure of the existing Janus software

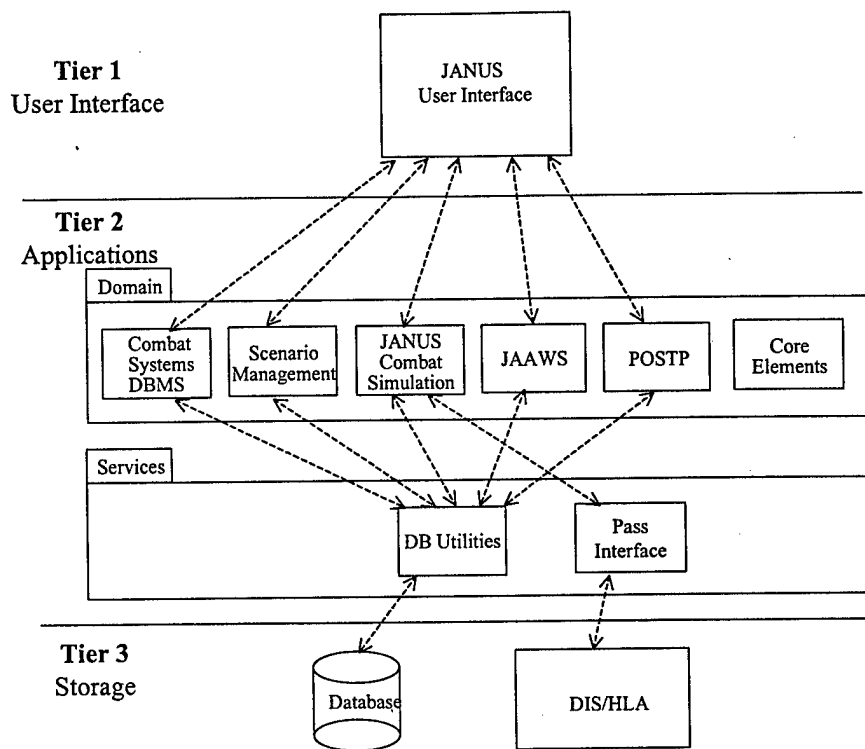


Figure 2. The proposed 3-tier object-oriented architecture.

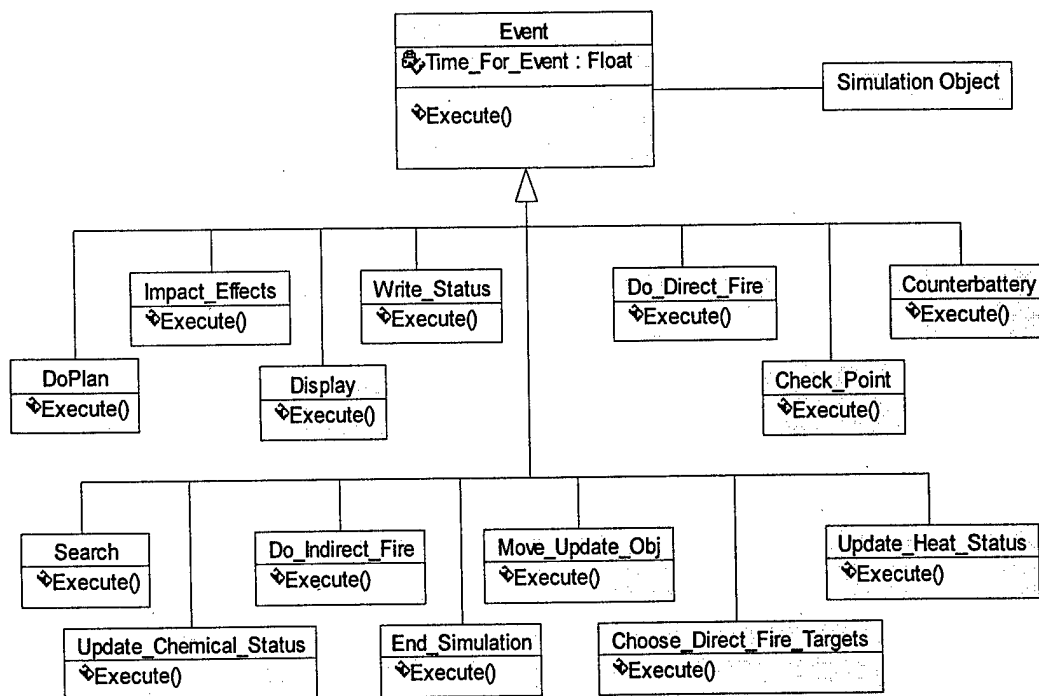


Figure 3. The event class hierarchy showing the different event handling operations of the JANUS Combat Simulation Subsystem.

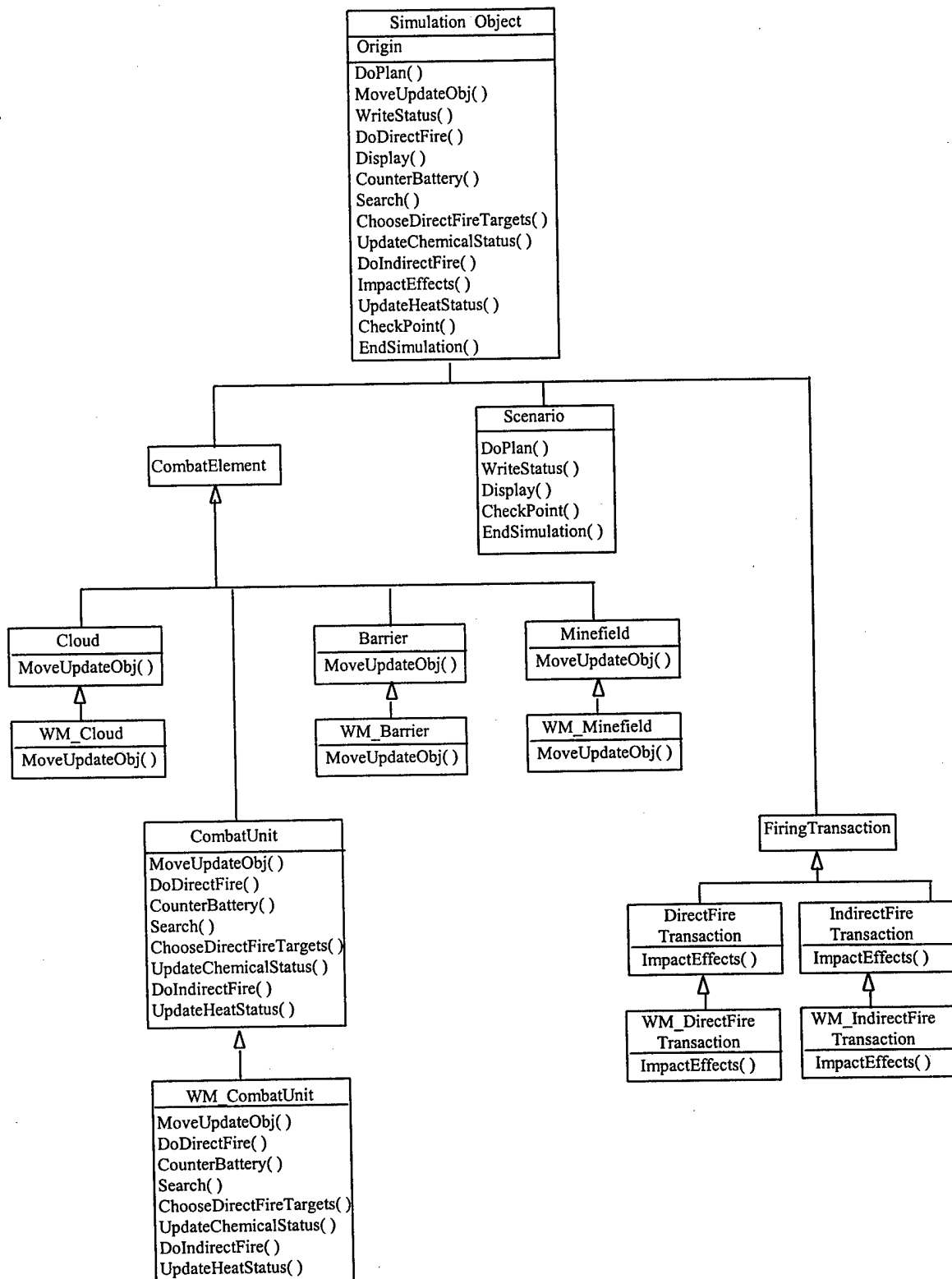


Figure 4. The simulation object class hierarchy showing the distribution of the event operations.

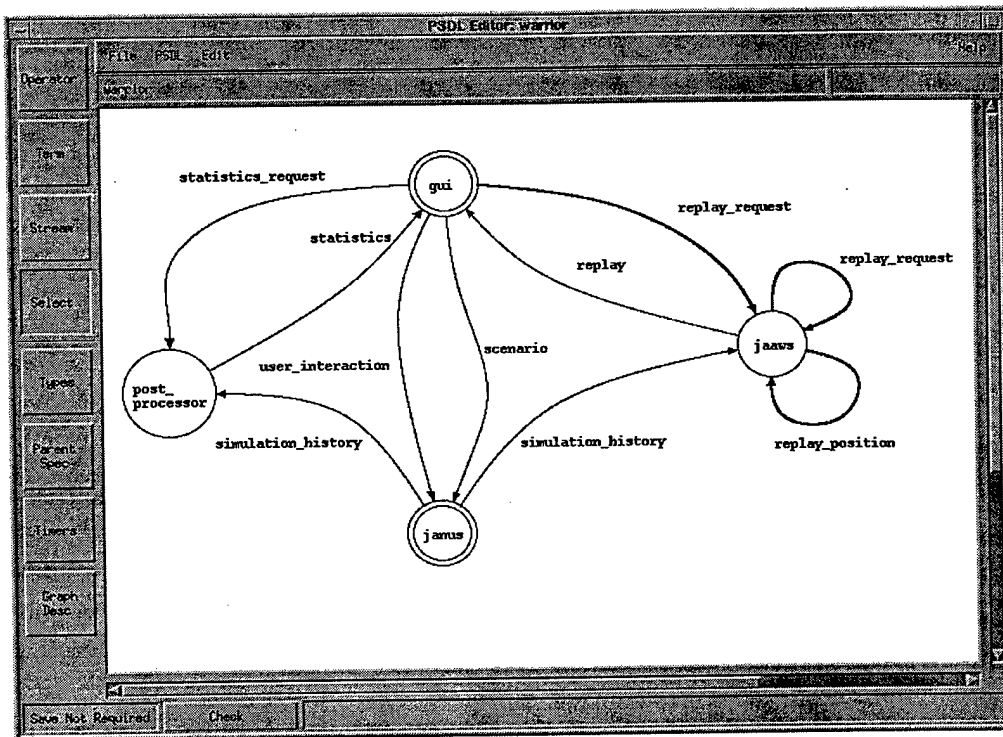


Figure 5. Top-level decomposition of the executable prototype

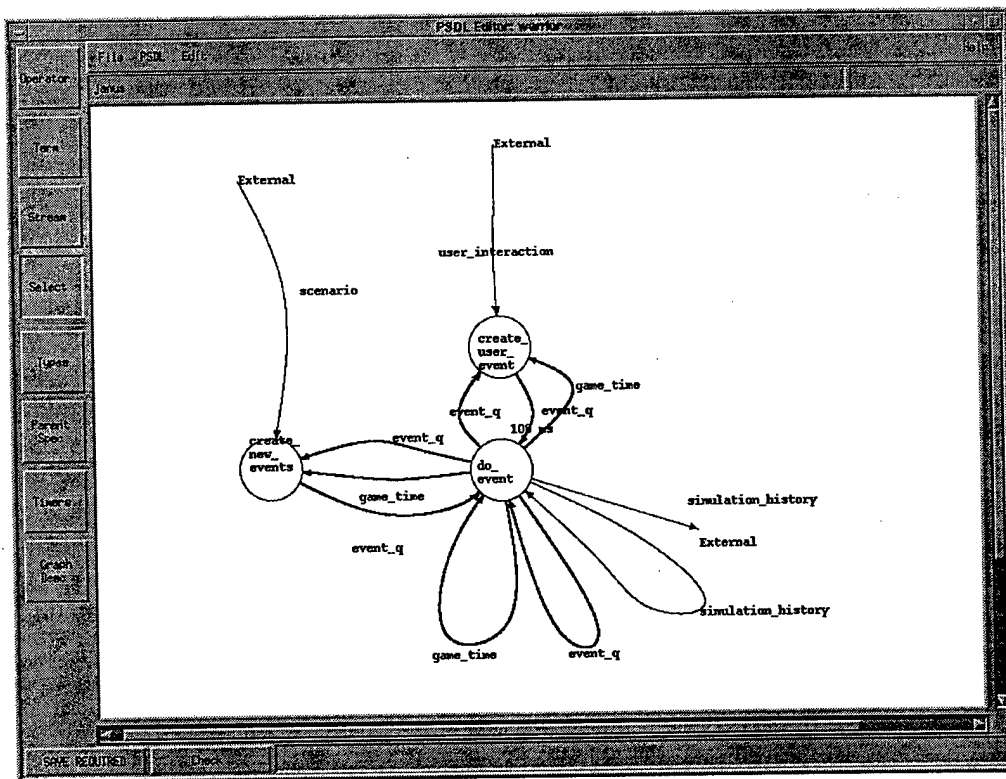


Figure 6. The JANUS subsystem of the executable prototype

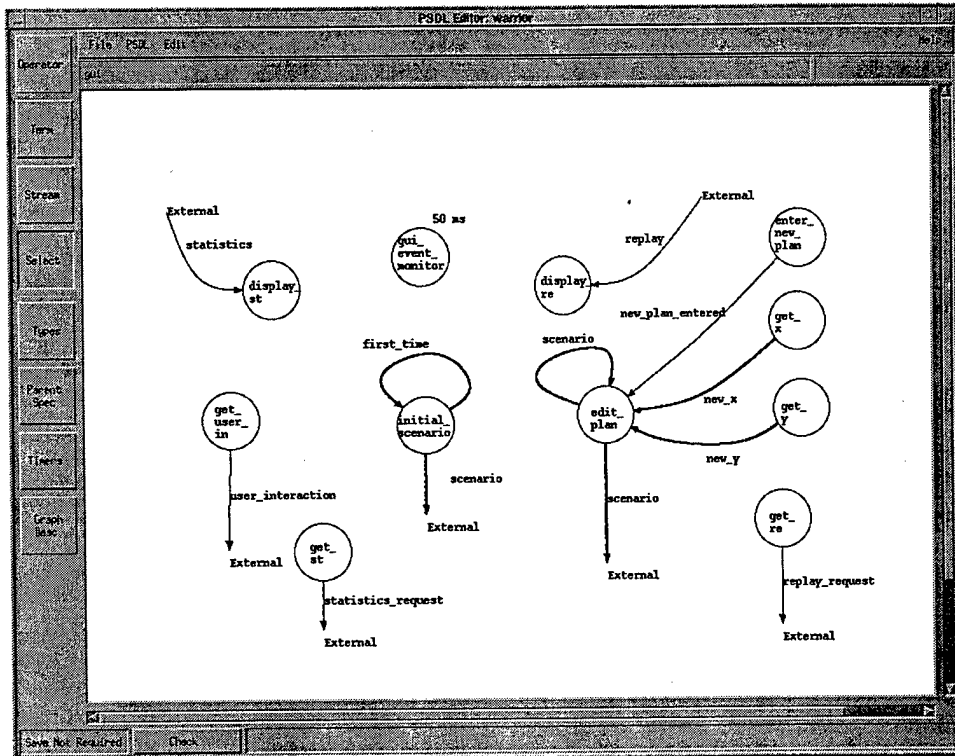


Figure 7. The GUI subsystem of the executable prototype

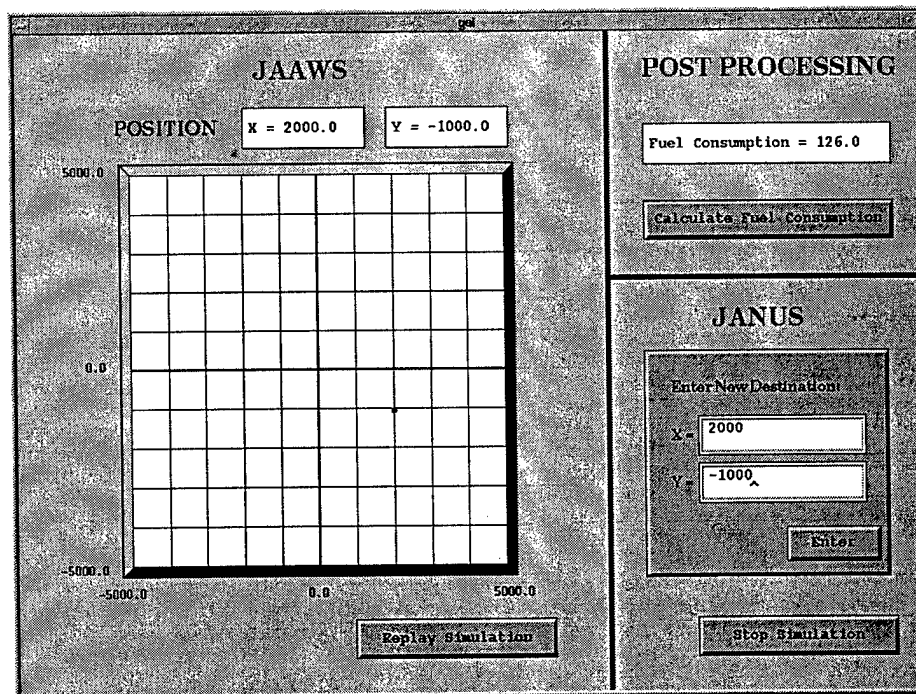


Figure 8. The Graphical User Interface of the executable prototype

Data Transmission Over Fiber Optics Using High Performance Network Protocol

John Drummond

Space and Naval Warfare Systems Center, San Diego
drummond@spawar.navy.mil

Abstract

The application of multimedia transmission has as one of its predominate components network communications. I have endeavored to implement and analyze network communication of multimedia information. In this analysis Xpress Transfer Protocol and Fiber Distributed Data Interface protocol have been utilized. This combination has provided a high speed and high performance network solution to the multimedia transmission that requires high bandwidth. FDDI is an implementation of the ISO standard for Open Systems Interconnect (OSI) Reference Model Data-Link and Physical layer protocols. These protocol implementations afford us a signaling rate of 100 Mbits/sec in addition to fault tolerance. XTP is an implementation of both the Transport and Network layers within the OSI Reference Model. The basis of this XTP implementations design is for high performance and efficiency. Employing this protocol I have successfully implemented both Unicast and Multicast network communication modes. The results of these implementations are presented in this paper.

Introduction

Multimedia (voice, video, data, text, and graphics) distribution over high speed networks has many commercial applications which has revolutionize the way computers and networks are used. Numerous commercial entities have formed strategic alliances to explore new opportunities in this area. I have been experimenting for several years with high speed networks which utilize FDDI, and a high performance network protocol called Xpress Transfer Protocol. I have performed many voice/video transmission experiments using XTP with several machines connected via FDDI network. The results indicate voice and video transmission using XTP and FDDI has many advantages over traditional methods of transmission including fault tolerance, high bandwidth, and data integration.

Xpress Transport Protocol

Xpress Transport Protocol is an implementation of the Open Systems Interconnect (OSI) Reference Model layers 3 and 4, or Network and Transport respectively. XTP allows for extensive application control over the network interface. Within the design of XTP full advantage is taken of the low Bit Error Rate (BER) offered by current network media such as fiber optics. This reduction in error processing allows a shift to the focus of completing other tasks.

XTP notability also includes improvements over other protocols[1]: Data Pipeline Size; Rate Control; Priorities; Out-of-band Data; No-error Mode; Policy Vs Mechanism; Multicast; Header/Trailer Protocol; Fixed-length Fields; Efficient Connection Setup and Teardown; Address Translation Routing; Retransmission; Acknowledged Control; Alignment; and VLSI Implementation. When compared to traditional protocols XTP offers various functional enhancements. XTP provides a pipeline design which can accompany megabit and gigabit networks (easily upgraded to terabit). Rate and burst control are accessible via parameters with the rate bits capable of being controlled at the receive side for synchronization purposes. Prioritization provides for user data discrimination. Policy vs mechanism being a primary design philosophy behind XTP allows user level control of implementation instead of fixed pre-established policies, an example of this is the no-error mode setting. Multicast also provides a functional enhancement over common protocols by providing semi-reliability of the network session. Additionally XTP has added features which focus upon performance. The addition of a header/trailer instead of the traditional header in a data packet allows for checksum to be calculated and appended thus reducing one sequential data pass. Fixed-length fields (i.e. header/trailer and flags) provides added efficiency. Connection setup/teardown for reliable transmission requires only two packets (vs TP4 six packets) in addition to providing various connection release paradigm support. XTP addressing translation provides for interoperability of Internet Protocol (IP) and ISO 8348 as well as many other network addressing designs. As XTP has implemented the Network layer protocol it supports routing also by bridging the

Transport/Network layers is has established a "transfer layer" architecture this routing has added efficiency. XTP provides selective retransmission when reliable-service/error-detection mode is desired. Elective acknowledgment control is also provided to the user by XTP.

Recent work within the XTP community is focused upon integration with the IP which resides on the OSI Reference Model Network layer. This number 3 layer provides the necessary routing between network segments. The integration of XTP with IP would greatly enhance XTP's implementation capabilities. Given the numerous IP based routers currently in use around the world this link-up between the two protocols would provide obvious advantages. The utilization of an enhanced Transport layer protocol such as XTP in combination with the popularly employed IP Network layer protocol will allow application specific network communication over Wide Area Network (WAN).

Fiber Distributed Data Interface

The Fiber Distributed Data Interface (FDDI) is an ANSI standard based on timed-token protocol technology. Within the OSI Reference Model FDDI implements a version of Physical and Data Link layers 1&2. A typical network consists of nodes connected by two fiber cables with a logical token circulating among the nodes and a signaling rate of 100 Mbits/sec. The FDDI architecture is to varying degrees fault tolerant. The network topology in our testbed will remain operational if a single fault, such as a cable break, occurs. The FDDI features that are useful in the transmission of voice and video are: High bandwidth (100 Mbit/sec); Very low error rates (10^{-9} BER); Predictable token access (low jitter); Large packet size (4500 bytes). Other characteristics of FDDI that serve practical purposes are: Fault tolerance; No electromagnetic emissions/interference; and Notion of priority.

Testbed Experiments

Our testbed consists of several commercially available Intel based PCs containing off-the-shelf components. Each network node PC is populated with Dual Attached Station FDDI cards, 10-bit resolution audio analog-digital converter cards using Adaptive Differential Pulse Code Modulation (ADPCM) audio compression with 16 kHz sample rate, and 2-card set of video interface boards consisting of JPEG video compression hardware and frame grabber engine which perform the digital video capturing and compression functions. All PCs are connected by a dual ring fiber optic cable.

The experiments follow two basic sequences of operation. The primary operation occurs on the transmitting side of the network communication connection. First task is to obtain video or audio analog data and to perform analog-to-digital conversion. Once completed this digitized information is then compressed using the appropriate data compression algorithm with the respective VLSI chipset (ADPCM for audio and JPEG for video) residing on the hardware. The compressed data is then formed into XTP network packets, by the application and XTP software, for transmission over the Fiber Optic network via the FDDI hardware in the node. The second set of operations is performed at the receiving node, beginning with the receipt by the Fiber Optic network and FDDI hardware in the node. These FDDI frames are processed by XTP and the resulting compressed data is delivered to the appropriate hardware for decompression and the resulting data is output to either the screen in the case of video data or the speaker in the case of audio data. This completes the communication cycle.

Audio Transmission

During the course of our audio experiments latency measurements were taken periodically. This measured latency was indicative of the time required for end to end (i.e. microphone-speaker) transmission. The results indicated a latency of approximately 25ms, which was well within the tolerable limits of perception by the human ear.

One-Way Delay	Effect of delay
>600ms	Incoherent
600ms	Barely Coherent
250ms	Annoying
100ms	Imperceptible (without network/original sample comparison)
50ms	Imperceptible (with network/original sample comparison)

Table 1: Effects of latency on human ear perception[3]

The latency tests were based upon XTP unicast mode communication link sending XTP network packets sized at 50 bytes each, and A/D converter buffered by an array of bytes 1024 long. This schema provided a good basis for testing and analysis. The ADPCM audio compression algorithm compresses the sampled audio waveform to 4 bits thereby reducing the data size by over 50% compared to 10 bit PCM digitization. This compression allows for very low network bandwidth utilization. When operating in unicast mode, the average consumption of network bandwidth given a typical compressed audio packet is .5035 Mbits/sec. XTP unicast mode is a network communication utilizing 2 nodes, where one node acts as a transmitter and another node acts as a receiver. This bandwidth is increased to approximately 1.102 Mbits/sec when utilizing duplex mode communication. XTP duplex mode involves two nodes and each node acts as transmitter and receiver simultaneously.

Video Transmission

The realtime video originated from various sources such as: Cable News Network (CNN) broadcast acquired from satellite downlink; Video Camera; and Video Tape. These sources all followed the NTSC format and all were fed into the video frame grabber. Again, during our experiments, latency measurements were recorded periodically. This time the measured latency was representative of the time required for picture to picture (i.e. screen display to screen display) transmission. The outcome of these tests revealed a latency of approximately 50-60ms (less than 2 frames) given that our experiments were based upon utilizing NTSC standard input which is 30 fps. This small latency is very difficult to perceive, even with the source and destination display screens side by side. Table 2 represents some results of a study [3] on frame rates and their effects on human eyes. As can be seen, a jerky motion is perceived when successive frames are 67-83 ms apart. This is in excess of the latency in our experiment between a frame appearing on the source screen and the same frame appearing on the destination screen.

Frames per second	Effect on human eye
<10 fps	Frames appear disjoint
12-15 fps	Motion is jerky.
30 fps	Television quality
60-75 fps	High-motion discernible (HDTV)
90 fps	Limit of human eye perception

Table 2: *Effects of frame rate on human eye perception*[3]

These latency tests were also based upon XTP unicast mode communication link with XTP network packets sized at 3305 bytes each, and a video buffer of 16000 bytes. The JPEG video compression chipset utilizing the Huffman encoding scheme provided us with 2 to 4 times data reduction thus greatly reducing the network bandwidth requirements for our realtime video communication experiments. A typical XTP unicast communication session utilized approximately 3 Mbits/sec to 6 Mbits/sec bandwidth. The XTP multicast sessions were recorded to also within the 3 Mbits/sec to 6 Mbits/sec range of network bandwidth consumption.

Conclusion

After performing the audio and video experiments in our testbed, the results have indicated the use of XTP and FDDI on multimedia transmission is feasible and may provide a bridge to the giga-bit network rates. The results of the XTP multicast bandwidth consumption is revealing in that it is within the range of the typical XTP unicast network utilization despite the fact that unicast is a 1 to 1 session and multicast is 1 to N network communication. As multimedia is becoming an important industry today, more research in multimedia transmission is needed.

References

- [1] W. Timothy Strayer, Bert Dempsey, Alfred Weaver, "XTP: The Xpress Transfer Protocol," Addison-Wesley Publishing Co, Inc., 1992.
- [2] "XTP Protocol Definition," Protocol Engines Inc. Report PEI 92-10, 1992.
- [3] Amit Shah, Don Staddon, Izhak Rubin, Aleksandar Ratkovic, "Multimedia Over FDDI," IEEE Proceedings, Sept, 1992, pp 13-15

Initial Distribution List

- | | | |
|----|---|----|
| 1. | Defense Technical Information Center
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, VA 22060-6218 | 2 |
| 2. | Dudley Knox Library, Code 52
Naval Postgraduate School
Monterey, CA 93943-5100 | 2 |
| 3. | Research Office, Code 09
Naval Postgraduate School
Monterey, CA 93943-5000 | 1 |
| 4. | David Hislop
US Army Research Office
Mathematical/Computer Science Department
4300 South Miami Blvd.
Research Triangle Park, NC 27709-2211, USA | 10 |
| 5. | Frank Anger
National Science Foundation
4201 Wilson Blvd.
Arlington, VA 22230 | 10 |
| 6. | Ralph Wachter
Office of Naval Research
Computer Science Division
800 N. Quincy St./ONR Code 311
Arlington, VA 22217-5660 | 10 |
| 7. | Ref Delgado
Defense Advanced Research Projects Agency
3701 N. Fairfax Drive
Arlington, VA 22203-1714 | 10 |
| 8. | Luqi, CS/Lq
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943 | 70 |

- | | | |
|-----|---|---|
| 9. | Astesiano, Egidio
Univ. Di Genova
director@disi.unige.it | 1 |
| 10. | Berry, Dan
Technion/Univ. of Waterloo
dberry@cs.technion.ac.il | 1 |
| 11. | Berzins, Valdis
Naval Postgraduate School
berzins@cs.nps.navy.mil | 1 |
| 12. | Bjorner, Nikolaj
Stanford University
nikolaj@cs.stanford.edu | 1 |
| 13. | Broy, Manfred
Tech. Univ. of Munich
broy@informatik.tu-muenchen.de | 1 |
| 14. | Carver, Doris
Louisiana State University
carver@bit.csc.lsu.edu | 1 |
| 15. | Cleaveland, Rance
SUNY, Stony Brook
rance@science.csc.ncsu.edu | 1 |
| 16. | Cooke, Dan
University of Texas at El Paso
dcooke@cs.utep.edu | 1 |
| 17. | Dabose, Mike
SPAWAR
dabose@spawar.navy.mil | 1 |
| 18. | Devanbu, Premkumar
University of California, Davis
devanbu@cs.ucdavis.edu | 1 |
| 19. | Drumond, John
SPAWAR
jd@spawar.navy.mil | 1 |

- | | | |
|-----|--|---|
| 20. | Ehrich, H.D.
Univ. Braunschweig
hd.ehrich@tu-bs.de | 1 |
| 21. | Evans, John
SPAWAR
evansjr@spawar.navy.mil | 1 |
| 22. | Gelfond, Michael
University of Texas at El Paso
mgelfond@cs.utep.edu | 1 |
| 23. | Green, Cordell
KESTREL
green@kestrel.edu | 1 |
| 24. | Gill, Helen
DARPA
hgill@darpa.mil | 1 |
| 25. | Guo, Jiang
Naval Postgraduate School
gj@cs.nps.navy.mil | 1 |
| 26. | Harn, Meng-chyi
Naval Postgraduate School
harn@cs.nps.navy.mil | 1 |
| 27. | Iyer, Purush
North Carolina State University
iyer@csc.ncsu.edu | 1 |
| 28. | Kraemer, Bernd
Fern Univ. Hagen
bernd.kraemer@fernuni-hagen.de | 1 |
| 29. | Lange, Doug
SPAWAR
dlange@spawar.navy.mil | 1 |
| 30. | Lee, Insup
University of Penn
lee@cis.upenn.edu | 1 |

- | | | |
|-----|---|---|
| 31. | Levitt, Raymond
Stanford University
rel@cive.stanford.edu | 1 |
| 32. | Lipton, James
Wesleyan University
lipton@wesleyan.edu | 1 |
| 33. | Liu, Junbo
KESTREL
liu@kestrel.edu | 1 |
| 34. | Mislove, Mike
Tulane University
mwm@math.tulane.edu | 1 |
| 35. | Polak, Wolfgang
wp@pocs.com | 1 |
| 36. | Pratt, Vaughan
Stanford Univ.
pratt@cs.stanford.edu | 1 |
| 37. | Ray, Bill
SPAWAR
ray@spawar.navy.mil | 1 |
| 38. | Robertson, Dave
University of Edinburgh
dr@dai.ed.ac.uk | 1 |
| 39. | Shatz, Sol
Univ. Illinois, Chicago
shatz@eecs.uic.edu | 1 |
| 40. | Shaw, Alan
Washington Univ.
shaw@cs.washington.edu | 1 |
| 41. | Shen, Lydia
SPAWAR
shen@spawar.navy.mil | 1 |
| 42. | Shing, Man-Tak | 1 |

Naval Postgraduate School
mantak@cs.nps.navy.mil

- | | | |
|-----|---|---|
| 43. | Uribe, Tomas
Stanford University
uribe@cs.stanford.edu | 1 |
| 44. | Waldinger, Rich
SRI
waldinger@ai.sri.com | 1 |
| 45. | Wegner, Peter
Brown University
pw@cs.brown.edu | 1 |
| 46. | Wirsing, Martin
Univ. Muenchen
wirsing@informatik.uni-muenchen.de | 1 |
| 47. | Yehudai, Amiram
Tel Aviv Univ.
amiram@math.tau.ac.il | 1 |
| 48. | Zhang, Du
Cal State University of Sacramento
zhangd@ecs.csus.edu | 1 |
| 49. | Zhao, Feng
Xerox PARC
zhao@parc.xerox.com | 1 |