1994-09

Proceedings of the 1994 Monterey Workshop, Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Evolution Control for Large Software

Proceedings of the

1994 Monterey Workshop

Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development:

Evolution Control for Large Software Systems
Techniques for Integrating Software Development Environments

Sponsored by:

Office of Naval Research
Advanced Research Projects Agency
Air Force Office of Scientific Research
Army Research Office
Naval Postgraduate School
National Science Foundation

September 7 - 9, 1994

U.S. Naval Postgraduate School
Monterey, California

Workshop Chairs

Luqi, Jim Brockett: Naval Postgraduate School

APL8176

Workshop Chairs

Luqi, Jim Brockett:
Naval Postgraduate School

Program Committee

Daniel Berry: Technion, Israel
Valdis Berzins, Mantak Shing: Naval Postgraduate School
Helen Gill: National Science Foundation
Joseph Goguen: Oxford University
David Hislop, Jagdish Chandra: Army Research Office
David Luginbuhl: Air Force Office of Scientific Research
John Salasin: Advanced Research Projects Agency
Ralph Wachter, Andre van Tilborg: Office of Naval Research

Local Arrangements

Mehdi Rowshanaee, Mauricio Cordeiro, Frank Palazzo,
Osman Ibrahim, Doan Nguyen: Naval Postgraduate School
David Dampier: Army Research Office
Salah Badr: Egyptian Armament Authority

Attendee List

Salah Badr
Egyptian Armament Authority
Cairo, Egypt

Jim Baker
Lockheed Research Center
baker@lmsc.lockheed.com

Daniel Berry
Technion, Israel
dberry@cs.Technion.AC.IL

Valdis Berzins
Naval Postgraduate School
berzins@cs.nps.navy.mil

Jim Brockett
Naval Postgraduate School
brockett@cs.nps.navy.mil

Daniel Cooke
University of Texas, El Paso
dcooke@cs.utep.edu

Mauricio Cordeiro
Naval Postgraduate School
cordeiro@cs.nps.navy.mil

Dan Craigen
ORA, Canada
dan@ora.on.ca

David Dampier
Army Research Laboratory
dampier@airmics.gatech.edu

Martin Feather
USC / Information Sciences Institute
feather@isi.edu

Joseph Goguen
Oxford University, UK
goguen@comlab.ox.ac.uk

Dave Hislop
ARO
hislop@aro-emh1.army.mil

Jim Hook
Oregon Graduate Institute of
Science & Technology
hook@cse.ogi.edu

Deepak Kapur
State University of New York at Albany
kapur@cs.albany.edu

Richard Kieburtz
Oregon Graduate Institute of
Science & Technology
dick@cse.ogi.edu

Mark Kindl
Army Research Laboratory
kindl@airmics.gatech.edu

Michael Lowry
NASA Ames Research Laboratory
lowry@ptolemy.arc.nasa.gov

Luqi
Naval Postgraduate School
luqi@cs.nps.navy.mil

Zohar Manna
Stanford University
zm@Theory.Stanford.EDU

Barbara Meyers
IBM Santa Teresa Laboratory
bfmeyers@VNET.IBM.COM

Raymond Paul
Operational Test and Evaluation Command
U.S. Army

Ted Ralston
ORA, Canada
ted@oracorp.com

C.V. Ramamoorthy
University of California, Berkeley
ram@cs.berkeley.edu

Bala Ramesh
Naval Postgraduate School
ramesh@nps.navy.mil

Ace Roberts
U.S. Army Missile Command
Redstone Arsenal
aroberts@sed.redstone.army.mil

David Robertson
University of Edinburgh
dr@aisb.edinburgh.ac.uk

John Salasin
ARPA
salasin@arpa.mil

Jack Schwartz
Courant Institute
New York University
schwartz@cs.nyu.edu

Richard Schwartz
Borland International
0003301326@mcimail.com

Alan Shaw
University of Washington
shaw@cs.washington.edu

Rob Shaw
University of California, Davis
shaw@cs.ucdavis.edu

Mantak Shing
Naval Postgraduate School
mantak@cs.nps.navy.mil

Steve Vestal
Honeywell Technology Center
vestal@src.honeywell.com

Dennis Volpano
Naval Postgraduate School
volpano@cs.nps.navy.mil

Richard Waldinger
SRI International
waldinger@ai.sri.com

Gio Wiederhold
Stanford University
gio@ARPA.MIL

Hongji Yang
De Montfort University, UK
hjy@de-montfort.ac.uk

Table of Contents

"Monterey Workshop '94: Software Evolution",
Luqi, Naval Postgraduate School 1

"Formal Support For Software Evolution",
Luqi, NPS, Joseph Goguen, Oxford University, UK, and Valdis Berzins, NPS 10

"Whither Formal Methods?",
Daniel Berry, Technion, Israel 22

"Software Change-Merging in Dynamic Evolution",
David Dampier, ARL, and Valdis Berzins, Naval Postgraduate School 38

"AMPHION: Towards Kinder, Gentler Formal Methods",
Richard Waldinger and Michael Lowry, NASA Ames Research Center 42

"Design of Languages for Multimedia Applications Development",
Jacob Schwartz and Kirk Snyder, New York University 46

"An Algebra for Ontology Composition",
Gio Wiederhold, Stanford University 56

"Making Specification Design More Accountable",
Dave Robertson and Jane Hesketh, University of Edinburgh 62

"Formal Methods for Complex Evolving Systems",
Steve Vestal, Honeywell Technology Center 69

"Formal Methods Technology Transfer: Impediments and Innovation",
Dan Craigen and Ted Ralston, ORA, Canada 74

"Real-Time Software and Proof Architectures",
Pam Binns, Honeywell Technology Center 85

"A Formal Model of Problem Solving and its Impact on Software Development",
Daniel Cooke, University of Texas at El Paso 90

"Feature-Oriented Software Engineering",
Herwig Egghart and Edgar Knapp, Purdue University 95

"Requirements Monitoring for System Maintenance in Dynamic Environments",
Stephen Fickas, University of Oregon, and Martin Feather, USC / ISI 107

"Formal Methods Experience and Recommendations",
Formal Methods Group, Fort George G. Mead, DoD 112

"Analytical Development of Control-Intensive Software",
R. Hardin, Z. Har'El and R. Kurshan, AT&T Bell Laboratories 116

"Generating Software from Specifications",
Richard Kieburtz, Oregon Graduate Institute of Science and Technology 122

"STeP: The Stanford Temporal Prover",
Zohar Manna, Stanford University 129

"Representing Rationale in Formal Systems Development",
Balasubramaniam Ramesh, Naval Postgraduate School 134

"Reengineering Real-Time Embedded Computer Software With the McCabe Tools",
Ace Roberts, U.S. Army Missile Command, Redstone Arsenal 139

"Temporal State Machines and Assertions: A Practical Framework for Handling Changes in Real-Time Systems",
Alan Shaw, University of Washington 145

"Towards A Practical Verification Environment for Concurrent Programs",
Robert Shaw, Ronald Olsson and Cui Zhang, University of California, Davis 150

"Formal Methods and Software Maintenance",
Hongji Yang, De Montfort University, UK 155

"A Software Evolution Control Model",
Salah Badr, Egyptian Armament Authority, Valdis Berzins, NPS 160

Monterey Workshop '94: Software Evolution — Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development

Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

1. Software and Formal Methods

We need effective and reliable ways to develop software systems that meet user needs. This will have a great impact on society; for example, DoD spends billions of dollars each year on software, but many software systems in use do not satisfy users' needs. An article entitled "Software Chronic Crisis" by W. Gibbs in the September 1994 issue of *Scientific American*, states "despite 50 years of progress, the software industry remains years — perhaps decades - short of the mature engineering discipline needed to meet the demands of an information-age society".

To remedy this situation, formal software models that can be mechanically processed can provide a sound basis for building and integrating tools that produce software faster, cheaper, and more reliably. Formal methods can also increase automation and decrease inconsistency in software development. For this purpose, researchers in formal methods need to better understand where the developers of large software systems need help. Also software developers need to better understand the benefits of using formal models to construct tools that can solve practical problems.

This year's workshop focuses on software evolution. This refers to all the activities that change a software system, as well as the relationships between those activities. This includes responses to requirements changes, improvements to performance or clarity, repairs of bugs, and the overall organization of the development process. Evolution is not just another name for maintenance; evolution occurs throughout the life cycle, and includes specification-based development, incremental/phased development, requirements prototyping, version and configuration control, on-line documentation, testing, code generation, etc. It captures the dynamic aspects of software development.

This workshop helps clarify what good formal methods are and what are their limits. According to Webster's Dictionary, *formal* means definite, orderly, and methodical; it does not necessarily entail logic or proofs or correctness. Everything that computers do is formal in the sense that syntactic structures are manipulated according to definite rules. Formal methods are syntactic in essence and semantic in purpose. Given the motivations of the workshop, we believe this is the most appropriate sense for the word "formal" in the phrase "formal methods."

The prototypical example of a formal system is first order logic. This system encodes first order model theory with certain formal rules of deduction that are provably sound and complete. Unfortunately, theorem provers for this system can be difficult to work with. Formal systems can also capture higher levels of meaning, e.g., for expressing requirements, but these systems are harder to work with and have fewer nice properties. Fine-grained formal methods have a mathematical basis at the level of individual statements and small programs, but rapidly hit a complexity barrier when programs get large. Formal support for large-grain aspects of software development is necessary to extend research results to large problems.

Software research cannot rely solely on the weak validation of proving theorems and checking logical consistency. The value of a contribution has to be measured by its impact on practical software development, rather than on how well it fits with existing mathematics or currently dominant trends in theoretical computer science. At present, we have no accurate formal models of software evolution and its relation to internal software structures. Existing models either focus on narrow aspects of the process or cover the entire life cycle through informal approaches that cannot be automated.

Formal methods can be very effective if the right method is applied to the right problem. The excessive optimism that everything important is provable helps to explain the excessive pessimism that nothing important is provable. Formal methods are often considered useful for proving that programs satisfy certain mathematical properties, but are also often considered too expensive to be practical. This view is narrow and limited because formal methods can reduce time to market, provide better documentation, improve communication, facilitate maintenance, and organize activities throughout the life cycle. We seek to clarify the conditions under which these benefits can best be realized.

Although there are formal models for several aspects of the software development process, each aspect has been considered in isolation, and effects that span the entire process are missing from the models. Such global dependencies are significant in software development, and the difficulty of maintaining these dependencies as software evolves is a limiting factor for large systems. We need to better understand the interactions among different parts of the process, and develop compatible models so that solutions to different parts of the problem can be combined into systems that span the entire process.

2. Increasing the Practical Impact of Formal Methods

Formal methods and tools do not yet adequately support the development of large and complex systems. The Monterey Workshop has sought to make progress by focusing on a specific topic related to formal methods each year. The first (1992) Monterey Workshop focused on real-time and concurrent systems. The second (1993) Monterey Workshop focused on software slicing and merging. This year's Workshop focuses on software evolution because it is important, even though the subject is difficult and not well explored.

This workshop assesses the practical impact of formal methods and tools, identifies gaps between the capabilities of formal methods and the practical needs of software development, and defines appropriate research directions. The workshop provides an opportunity to share recent advances in formal methods and their integration in software development environments.

We need groups of people working together on different aspects of the software problem, on a long-term basis. Users need to learn the capabilities of formal methods and researchers need to learn the real issues in practical software development. It would be desirable to achieve a consensus on what are the most important aspects, and how they might fit together, supported by an on-going discussion as our understanding grows.

3. Software Evolution and Prototyping

Software evolution has been addressed from many viewpoints. These include management issues such as which parts of a system should be rebuilt, configuration issues such as keeping track of many versions of the same system, testing issues such as determining which test cases could be influenced by a given change, requirements issues such as determining when the assumptions underlying a system specification are no longer valid, code restructuring issues, performance improvement issues, and much more. Many of these problems do not have accepted or validated formal models, and the application of formal methods is less obvious than in more mature areas, such as proving program properties.

Software evolution includes managing all dynamic structures and activities involved in software development. Much of the work in software process modeling does not emphasize automation. This is because this area failed to separate management issues from computer support issues. Both aspects are needed to help managers make better decisions about software evolution with less effort. Software development processes cannot be totally automated, because human judgement is required for good management.

A traditional view is that software evolution only occurs after initial development is completed. For example, software evolution has been defined to consist of the activities required to keep a software system operational and responsive after it is accepted and placed into production. This term previously referred to maintenance activities, to avoid the deadly negative connotation of that word. Evolution has the connotation of life, and it captures the dynamic aspects of software development if used in the context of an alternative software life cycle like prototyping that includes all activities from requirements specification and system construction to updating operational systems.

Alan Perlis took a broad view of software evolution, emphasising the parallel with biological evolution[1]. Here software systems are analogous to biological species, which adapt themselves to survive in changing ecological niches. Perlis was also interested in the analogy with Maturana's notion of autopoietic systems, which continually construct themselves from their own parts in order to adapt.

In the design and development of our Computer Aided Prototyping System (CAPS), we learned the importance of automated support for software evolution. This project worked in parallel on formal modeling, developing software tools based on the models, and applying the tools to practical problems. We found significant synergy among these activities, changing our approach to all three. Many versions of models and tools have been developed and modified during this process. Our work[2] on formal modeling and experimentation for software evolution is summarized in my joint paper with Drs. Goguen and Berzins and the paper by Drs. Dampier and Berzins in the proceedings of this workshop.

4. Goals of the Workshop

This workshop involves three groups of people coorperating to understand and solve software evolution problems. These are software developers, theoreticians, and sponsors; they come

[1]This paragraph is based on the recollections of Joseph Goguen of some conversations with Perlis late in his life.

[2]This work has been supported in part by the National Science Foundation and the Army Research Office.

from industry, government, and academia. We have an opportunity to learn and to exchange technical information, including some that would not normally be published for economic and security reasons. For software researchers, nothing can be more important than a better understanding of what is needed for real world software development. This help to avoid research results that are only good for toy problems. Industry people can help this process by clearly explaining what their most important problems are, separating essential aspects from the incidental complications of particular situations, and separating technical problems from political and management difficulties.

We have people with strong backgrounds in many branches of mathematics. Hopefully we can narrow some of the problems so that appropriate theoretical models can be applied to important aspects of practical software development. We may not have the luck to find off-the-shelf mathematics solving our problems. Real problems often have scary complexity. We need help in dividing real problems into subproblems to reduce the complexity. It may also be necessary to approach problems in completely different ways, e.g., probabilistic algorithms can handle some problems of practical size that are intractable using deterministic methods. The software evolution problem is interesting and difficult. Hopefully this workshop will be the start of progress in the formal modeling of software evolution, in a way that will have significant impact on practical software development.

The workshop schedule was organized as follows:

Day 1: Introduction to software evolution & formal methods in software development
Jim Brockett, Naval Postgraduate School, Welcome and Introduction.
Luqi, Naval Postgraduate School, Monterey Workshop '94: Software Evolution.
Daniel Berry, Technion, Israel, Whither Formal Methods?
Mantak Shing, Jim Brockett, NPS, Computer-Aided Prototyping System (CAPS) Demonstration.
Barbara Meyers, Senior Programmer, IBM, Richard Schwartz, VP of R&D, Borland International, Industry Perspectives.
Valdis Berzins, Naval Postgraduate School, An Evolution Control Model.

Day 2: Evaluations of successful formal methods
David Dampier, Army Research Laboratory, Software Change-Merging in Dynamic Evolution.
Richard Waldinger, SRI International, Michael Lowry, NASA Ames Research Center, AMPHION: Towards Kinder, Gentler Formal Methods.
Jacob Schwartz, New York University, Design of Languages for Multimedia Applications Development.
Gio Wiederhold, Stanford University, An Algebra for Ontology Composition.

Day 3: Applications of formal methods and summary of the workshop
Steve Vestal, Honeywell Technology Center, Formal Methods for Complex Evolving Systems.
Dan Craigen and Ted Ralston, ORA, Canada, Formal Methods Technology Transfer Impediments.
Dave Robertson, University of Edinburgh, UK, Making Specification Design More Accountable.
Luqi, Naval Postgraduate School, Wrap-up Discussions.

5. Summary and Conclusions

This section summarizes and synthesizes the conclusions reached in discussions at the workshop, especially the final session. This is necessarily a creative task, because of divergent viewpoints among the participants. The presentations by Berry, Craigen and Ralston were particularly effective in stirring up controversy. Much of the discussion was about formal methods in general, rather than about their direct application to software evolution, although it always hovered in the background as a central motivation. Subsection 5.4 below focuses specifically on evolution.

5.1 A Vision for Formal Methods

The workshop generally agreed that there is now much more to formal methods than suggested by the themes dominant in the past, namely correctness proofs for algorithms and program synthesis. Although both of these remain interesting topics for theoretical research, workshop participants felt that their direct impact on the practice of large scale software development was limited.

Papers in the workshop suggest a vision for the future that is less ambitious and more realistic than that of the past. This new vision calls for using formal models and algorithms as a basis for computer tools that can help to solve practically significant problems; these problems are more limited and well defined than in the past. This approach gives up the unrealistic artificial intelligence goal of fully automating software development, by recognizing that human understanding and creativity must play an important role. It also recognizes that user requirements changes are a dominant aspect of practical software development, and gives up the ambitions of general-purpose program verification and synthesis. Past research in formal methods contributes to this new vision by providing the basic concepts and algorithms needed for building more practical models and tools.

All participants in the workshop seemed to agree that formal methods could be very useful, and indeed were crucial for making software engineering into a discipline that is as well understood and organized as other engineering disciplines, each of which relies on sound and well-tested mathematical models.

5.2 An Emerging Paradigm?

A number of successful applications of formal methods reported at the workshop seem to form a cluster suggesting an emerging paradigm for applying formal methods. These applications involve a tool having all or most of the following attributes:

1. They address a narrow, well defined, and well understood problem domain; there may even be an existing, successfully used library of program modules for the domain.

2. There is a coherent user community interested in the problem domain; users have some understanding of the domain, good communication among themselves, and potential financial resources.

3. The tool has a graphical user interface that is intuitive to the user community, embodying their own language and conventions.

4. The tool takes a large grain approach; rather than synthesizing procedures out of statements, it synthesizes systems out of modules; it may use a library of components and synthesize code for putting them together.

5. Inside the tool is a powerful engine that encapsulates formal methods concepts and/or algorithms; it may be a theorem prover or a code generator; users do not have to know how it works, or even that it is there.

Examples of systems described in the proceedings that fit this discussion are: CAPS (see the paper by Luqi, Goguen and Berzins); ControlH and MetaH (see the paper by Vestal); AMPHION (see the paper by Waldinger and Lowry); the Panel system (see the paper by Schwartz and Snyder); the work of Robertson and Hesketh; and the DSDL translator of Kieburtz. If we were to suggest a name for this emerging paradigm, it might be *Domain Specific Formal Methods*, in recognition of the role played by the user community and their specific domain. Generic formal architectures and tools for a variety of software application domains are promising for future research.

5.3 Problems and Limits of Formal Methods

There was a great deal of discussion in the workshop about problems with the current state of formal methods. It is not surprising that some of this discussion repeated ideas that have become common in the literature, but some of it seemed fresh and original.

Attendees noted that formal notation is alien to most practicing programmers; most have little training or skill in formal mathematics. This motivates points 3. and 5. in Section 5.2. Discussion suggested that this problem is worse in the U.S. than in Europe. For example, set theoretic notation is better accepted in Europe.

Another problem is that some advocates of formal methods take a highly dogmatic position, that absolutely everything must be proved, to the highest possible degree of mathematical rigor; it must at least be machine checked by a program that will not allow any errors or gaps, and preferably it should be produced by a machine. In discussion, it was noted that mathematicians hardly ever achieve, or even strive for, such rigor; published proofs in mathematics are highly informal, and often have small errors; they never explicitly call upon rules of inference from logic (unless they are proving something *about* such rules). There are various degrees of formality, and the most rigorous are very expensive; such efforts are only warranted for critical aspects of systems. Practitioners would like to be able to combine informal development of the bulk of an application with formal methods for the critical parts.

Another problem raised in the discussion is that formal methods tend to be rigid and inflexible; in particular, it is difficult to adapt a formal proof of one statement to prove another, slightly different statement. Unfortunately, in the world of practical programming, requirements and specifications are constantly changing, so that such adaptations are frequently necessary. But classical formal methods have difficulty in dealing with the changes that are such an important part of the real world.

Another problem is that formal methods papers and training often deal only with toy examples, and often these examples have been previously treated in other formal methods papers. Although it may not be possible to give a detailed treatment of a realistic example in a research paper or classroom, it is still necessary that such examples exist for a method to have credibility. To be effective, training in formal methods should treat some parts of a realistic (difficult) application. A related problem is that much of the research in formal methods is not applications oriented.

There are also technical difficulties with some formal methods. For example, the Z language is hard to use for proving properties of either the specification itself or of associated code. This is because the specification has to be flattened before it can be used, since the structure of its specifications is usually very different from the structure of associated proofs and code. In particular, its schemas do not encapsulate their content, and only support re-use at the text level. Proofs about Z specifications must use the axioms of set theory, and this can be very difficult. For example, proofs using algebra are easier, because the axioms and rules of inference are much simpler, as well as easier to automate. It is hard to convince software developers to use a formal method with technical deficiencies and esoteric symbols.

It is useful to distinguish among small, large, and huge grain methods. This distinction refers to the size of the atomic components that are used, rather than the size of the system itself. The "classic" formal methods fall into the small category. In particular, pre- and post- conditions, Hoare Axioms, weakest preconditions, predicate transformers and transformational programming all have small size atomic units, and fail to scale up. In general, these methods have difficulty handling changes to specifications and requirements, and thus have difficulty with maintenance and fit poorly with software evolution. Transformational programming is less sensitive to errors than the other methods, but has the particular problem that there is no bound to the number of transformations that may be needed; this restricts its use to relatively small and well understood domains.

Some of these methods also have technical deficiencies. For example, the original first order formulation of weakest preconditions is inadequate for expressing loop invariants, which are fundamental for software specification. Without loops and equivalent constructs, programming languages would be nearly useless. This problem has been largely ignored by the formal methods community. However, a second order formulation is adequate, and serves as the foundation of the specification language SPEC used in teaching and research on software engineering at the Naval Postgraduate School for many years.

5.4 Formal Methods for Software Evolution

This subsection discusses issues that relate specifically to software evolution. Software evolution is poorly understood at present, and therefore more in need of help from formal models and methods than many other areas of software engineering. It can be very difficult to formalize a new area, especially without understanding from sponsoring agencies.

The difficulties in this area are not purely technical; social, political and cultural factors are also important, and can dominate the cost of software development. Tools based on formal models can help with both technical and management tasks. They can maintain the integrity of a software development project by scheduling project tasks, monitoring deadlines, assigning tasks to programmers, keeping on-line documentation, maintaining relations among system components, tracking versions, variations and dependencies of components, and merging changes to programs. These problems are important when a large group of programmers work concurrently on a large complex system.

Several workshop participants mentioned requirements capture as an important problem: it is necessary to know what to build, but in fact, this is always a moving target for large complex systems. Constantly changing requirements are a major cause of the difficulty of building such systems; this phenomenon has been called *requirements drift*. An extreme form of this is *requirements inflation*, where the requirements grow so much that the system development effort collapses.

A related problem is *traceability*, which is the problem of tracing design decisions, or fragments of specification or code text, back to the requirements that they are supposed to meet. The core of this difficulty is maintaining a complex network of links against the constantly changing requirements, which in turn imply constantly changing specifications and code. Real development projects for large complex systems rarely even attempt this, and those that do find it excessively burdensome, since the current state of practice requires manual entry and update of all dependencies. Since the benefits of adequate tool support for traceability would be enormous, effective formal models for this aspect are of interest. Particular subproblems in this area are formalizing dependencies and developing methods for calculating dependencies and propagating the implications of a change through a dependency network.

Another aspect of this problem is the difficulty of maintaining the dependencies among components in a large software system development effort. Often the components are not adequately defined, e.g., module boundaries may be incorrectly drawn, or not even explicitly declared; also, interfaces may be poorly drawn or badly documented. Without formal models of the dependencies and tool support for managing them, it is impossible to know what effect a change to a component will have, and in particular, to know what other components may have to be changed to maintain consistency. Methods for supporting changes to module boundaries would be useful.

An important practical problem for industry is to deal with so-called "legacy code," that is, old code that is poorly structured and poorly documented; often, it is written in an obsolete or obscure language, and nearly always the programmers who wrote it are long gone. For example, many banks depend upon huge COBOL programs for the success of their enterprise, but find it extremely difficult to modify these programs when business conditions change. As pointed out by Jim Baker from Lockheed, Barbara Meyers from IBM, Richard Schwartz from Borland and Raymond Paul and Ace Roberts from the U.S. Army, software researchers have to accept the realities found in industry and the DoD, as this is the source of their scientific and economic impact.

5.5 Promising Directions

The papers and discussions in the workshop, and the summaries given above, suggest several directions that may be promising for future research.

The first of these was called Domain Specific Formal Methods in Section 5.1. Recall that this involves encapsulating formal models or algorithms, such as an inference engine or program generator, into a tool with an intuitive graphical user interface for writing programs for a specific application domain. The animation of formal languages can be a useful complement to this approach in practice, e.g., for debugging.

Several participants, including Profs. Ramamoorthy, Goguen, Kieburtz, and Shing, raised interesting points about teaching formal methods. The negative impact of teaching a formal method and ignoring the social, political and cultural problems that necessarily arise in real projects was mentioned. For example, students may be taught programming from formal specifications, but not that specifications come from requirements, and that requirements are always changing. As a result, they are not prepared for the rapid pace of evolution found in real industrial work. A related problem is that many students feel that formal methods turn programming from a creative activity into a boring formal exercise. This can cause them to leave the field. Students need to know how to deal with real programs having thousands or even millions of lines of code, and carefully crafted correctness proofs for simple algorithms give an entirely misleading impression of what real programming is like. Most of the techniques in textbooks and the classroom are small grain and do not scale up to large complex problems.

Reliable formal method based tools can let students do problems that would be impossible by hand; this should increase their confidence. Teachers should also present methods and tools that work on large grain units, that is, on modules, rather than on small grain units like statements, functions and procedures, because these scale up, whereas the small grain methods do not. It is desirable to develop suites of sample problems that systematically show how and when to apply formal methods, and how to combine them with informal approaches. These goals will require refining and extending existing formal methods and tools, developing more natural user interfaces, rethinking process models, revising curricula, retraining teachers, and experimentally validating the resulting methods in practical situations.

Some participants pointed out that many successful applications of formal methods have occurred in the hardware area. Hence this is a good demonstration of the value of formal methods. It also continues to be a good area for further automation and research.

The discussion outlined above emphasized the importance of applications for formal methods, suggesting a need-driven approach, as opposed to a topic-driven approach. Basic research in computational logic still provides the foundation for many practical applications of formal methods. It is important to avoid a short term view of what technology needs, and also to avoid overselling formal methods, either as a general field or as an approach to particular applications.

Taking a long term integrated view, formal methods are beginning to make a real impact on practical software development, and this impact is likely to increase. It seems unlikely that general purpose tools, such as theorem provers for first order logic, will have an immediate impact on users, although they can be useful inside more narrowly focused tools. Software evolution is a challenging but rewarding application area, where formal models are likely to have a large impact if they can be formulated for the right problems. This can be a major step in turning software engineering into a true engineering discipline, with a solid mathematical foundation on which to base its practice.

Acknowledgements

I would like to thank: the participants in the workshop; the co-chair Jim Brockett; the members of the program and local arrangements committees; and the sponsors, the Army, Navy and Air Force Research Offices, the National Science Foundation, and the Advanced Research Projects Office.

Formal Support for Software Evolution[1]

Luqi, J. Goguen, V. Berzins

1. Why Software Evolution

Software evolution refers to all the activities that change a software system, as well as the relationships among these activities. This includes responses to requirements changes, improvements to performance or clarity, repairs of bugs, and the overall organization of the software development process. The older term "maintenance" refers to these activities only after initial development, in the context of the traditional life cycle. Prototyping is the process of quickly building and evaluating a series of prototypes, where a prototype is a concrete executable model of selected aspects of a proposed system. In prototyping, evolution activities are interleaved with development, and continue after the delivery of the initial version of the system.

Current capabilities for software evolution are inadequate. The most visible symptoms of the problem are the large backlog of requested changes, long delays, high error rates, and an alarming incidence of failure to complete changes. Some causes of these problems are missing information, intellectual overload, and primitive tool support.

We need accurate information about requirements, design rationale, and dependencies in order to change and manage code at low cost [2]. This information is rarely recorded and kept up to date, in part due to the cost, effort levels, and lack of incentives. Although management has been reluctant to accept the cost of adequately supporting evolution, recent failures of large projects such as the baggage handling system for the Denver airport, which had $20 million of requirements changes long after construction had begun [5], and the recent emergence of installations that operate at the highest level of the SEI process maturity model, suggest that competitive pressures may soon leave no other alternative.

The data and dependencies in a large software system are so complex that unaided human understanding cannot cope with them effectively. Tool support is sparse, primitive, and mostly low-level, e.g., facilities for generating cross-reference listings and for editing and storing different versions of program documentation. While such tools can reduce the mechanical work involved, they do not provide the kind of support needed to handle the complexity of real software evolution problems reliably.

The capabilities of current tools for supporting software evolution are limited by lack of formal models and theoretical bases. Formal models can lead to tools that enhance the capabilities of people to reliably change complex software systems. Such tools will make it easier to formalize and record more of the dependencies in a design, and to provide more sophisticated kinds of decision support based on this information.

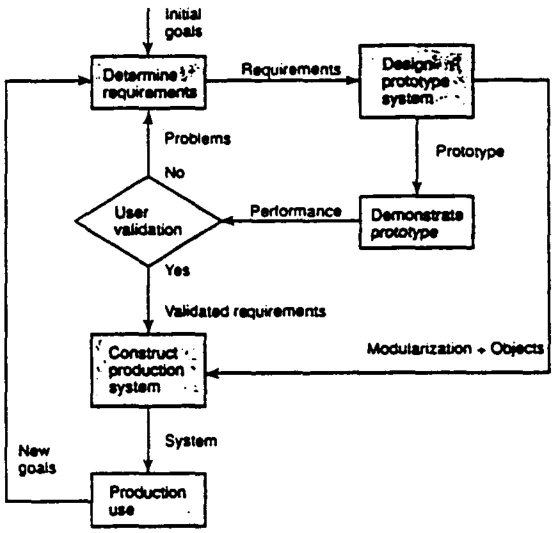
This paper describes formal models for software evolution and the automation support based on these models developed by Prof. Luqi's Computer-Aided Prototyping (CAPS)

[1] This research was supported in part by the National Science Foundation under grant number CCR-9058453, in part by the Army Research Office under grant number ARO-145-91, and in part by a grant from Fujitsu Laboratories Limited, Japan.

project at the U.S. Naval Postgraduate School. Section 2 describes the relation between software evolution and prototyping; a graph data model for evolution that records dependencies and supports automatic project planning, scheduling, and version management; a model of software changes that supports automatic change merging; a model of software behavior that supports automatic retrieval of reusable components; and a model of timing constraints that supports generation of schedule code. Section 3 presents some conclusions.

2. Software Evolution Through Computer-Aided Prototyping

Figure 1 illustrates the iterative prototyping cycle. The user and designer work together to define the requirements for critical parts of the envisioned system. The designer constructs a prototype at the specification level. Demonstrations of the prototype let the user evaluate the prototype's actual behavior against its expected behavior, identify problems, and work with the designer to redefine requirements. This process continues until the prototype successfully captures the critical aspects of the envisioned system. The designer then uses the validated requirements as a basis for the production software. In this way, software systems can be delivered incrementally and requirements analysis can continue throughout the system's lifetime. Incremental delivery lets users gain early experience with the software, leading to new goals, triggering further iterations, and extending the advantages of prototyping to the production environment.



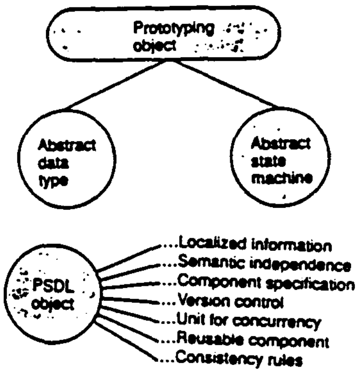


Fig. 1 The Prototyping Cycle Fig. 2 Class Structure and Properties of PSDL Objects

The problems of software evolution are especially prominent for rapid prototyping because prototypes are subject to frequent and repeated changes. Therefore computer support for evolution is essential for its practical success. Our computer-aided prototyping system CAPS and its prototyping language PSDL support software evolution. We have used formal models in several areas to provide automation support.

CAPS consists of an integrated set of tools that help design, translate and execute prototypes [11]. These include a graph data model for evolution, evolution control system, change merging facility, automatic generators for schedule and control code, and automated retrievals for reusable components.

The prototyping language PSDL provides a simple way to abstractly specify software systems for both prototypes and production software [10]. A PSDL program consists of two kinds of object, corresponding to abstract data types (PSDL types) and abstract state machines (PSDL operators); see Fig. 2. Their function is to localize the information for analyzing, executing and reusing independent objects. Objects are also the basis for version control, and are natural units of work in a distributed implementation.

2.1. A Graph Data Model and Control System for Evolution

We have developed a graph data model for evolution to provide computer aid for managing both the activities in a software development project and the products of those activities [12]. This data model represents the software system evolution history and plans as an acyclic bipartite graph $EG = (S, C, O, I, used\_by, part\_of)$. Evolution step nodes in S represent development *activities* and component nodes in C represent *products* of those activities. The output edges O link an evolution step to the components it produces and the input edges I link components to the evolution steps that use them. The graph also contains *part_of* edges representing the decomposition structure of composite steps and components, and *used_by* edges showing dependencies between components whose derivation history is not included in the graph. The latter are used to account for components produced outside the system that maintains the graph, such as legacy software.

Components are immutable versions of objects, and are labeled with unique object identifiers. The objects can be problem reports, change requests, reactions to prototype demonstrations, requirements, specifications, manuals, test data, design documents, program code or other items. The I and O edges capture derivation dependencies between the various versions of an object.

Two versions of the same object belong to the same *variation* if one was derived from the other. Variations are paths in the graph whose component nodes are all versions of the same object. Different variations of the same object represent parallel lines of development. The dependencies represent the essence of the evolution history.

The model associates a finite state machine with each step to represent the status of the step; the status states are proposed, approved, scheduled, in progress, completed, and abandoned. These states keep track of the boundary between the future and the past, and provide a means for managing the progress of the project.

The evolution control system based on this data model [1] provides algorithms that support project planning, scheduling, and design management. The system has been implemented using the Ontos object-oriented database. The support provided by the evolution control system includes automatic identification of induced steps implied by dependencies between components (e.g., if a requirement is modified then the program components derived from that requirement must also be modified). The set of induced steps can be defined formally as follows:

$$induced\_steps(s) = \{s' : S \mid \exists\, c, c' : C\ [\ c\ primary\_input\ s\ \&\ c\ affects\ c'$$
$$\&\ c'\ primary\_input\ s'\ \&\ c'\ in\_scope\ s\ \&\ current(c')\ ]\}$$

$$c\ primary\_input\ s \iff c\ I\ s\ \&\ \exists c' : C\ [\ s\ O\ c'\ \&\ same\_variation(c, c')\ ]$$

$$c\ in\_scope\ s \iff \exists c' : C\ [\ c'\ I\ top(s)\ \&\ c'\ affects\ c\ ]$$

$$current(c) \iff not\ \exists c' : C\ [\ c\ affects\ c'\ \&\ same\_variation(c, c')\ ]$$

$$same\_variation(c, c') \iff object\_id(c) = object\_id(c')\ \&\ (c\ affects\ c'\ or\ c'\ affects\ c)$$

where the function *top* is defined by the following:

$$\forall s : S\ [\ s\ part\_of^*\ top(s)\ \&\ not\ \exists s' : S\ [\ top(s)\ part\_of\ s'\ ]\ ]$$

The edge types I and O are used above to denote the corresponding binary relations on the vertices of the graph. The predicate *affects* is the transitive closure of the relation defined by the edges $(I \cup O \cup used\_by)$. The predicate *part_of** is the reflexive transitive closure of the relation defined by the *part_of* edges in the graph.

Support for planning includes a generated default decomposition for each proposed step from the decomposition of the current version of the affected components. The system constructs schedules and work assignments based on other attributes and relationships that reflect management decisions, such as deadlines and priorities of steps and skill levels of designers. The system provides up-to-date estimates of expected completion time on demand, alerts the project manager when situations arise that can impact project deadlines, and suggests deadline adjustments. It adjusts the schedule continually as more information becomes available, and uses the management policies recorded as attributes of steps to automatically assign new tasks to designers as they complete previous tasks.

Although the designer scheduling problem is NP-hard, we have found experimentally that a linear bound on the number of backtracking points allows a branch-and-bound algorithm to find schedules for all of the several hundred randomly generated feasible scheduling problems we have tried, while preventing impractical search times for infeasible problems. The system has been able to find schedules for up to 1200 tasks with practical running times. The bound on the search enables the system to detect infeasible deadlines with high probability and without excessive computation. This capability is used to automatically suggest nearly minimal deadline changes that will transform an infeasible scheduling problem into a feasible one.

The evolution control system also uses the dependency information contained in the project plans to deliver the proper versions of the required input components for each step to the designer responsible for the step and to insert the versions of components produced by completed steps in the proper places in the graph, thus completely automating the check in and check out procedures from the project design database.

2.2. A Hypergraph Data Model for Evolution

This section describes a more abstract version of the graph theoretic model for software evolution presented in [11,12] and further developed in [1]. All these models are *data models*, in the sense that they describe an abstract data type for keeping track of the versions and variations of the components and systems that arise during a system development effort. The model described here simplifies and clarifies the previous models by directly incorporating some of their features into a more abstract mathematical structure using hypergraphs. Hypergraphs generalize the usual notion of graph by allowing *hyperedges*, which may have multiple output nodes and multiple input nodes. Our intention is to make the software evolution model easier to modify and extend, as well as easier to understand and implement.

The following mathematical definitions are used in the software evolution model:

Definition: A (directed) **hypergraph** is a tuple (N, E, I, O) where

1. N is a set of **nodes**,
2. E is a set of **hyper edges**,
3. $I: E \rightarrow 2^N$ is a function giving the **inputs** of each hyperedge, and
4. $O: E \rightarrow 2^N$ is a function giving the **outputs** of each hyperedge.

A **path** p **from** a node n **to** a node n' is a sequence of $k > 0$ edges $e_1 ... e_k$ and a sequence $n_1, ..., n_{k+1}$ of nodes such that $n_i \in I(e_i)$ and $n_{i+1} \in O(e_i)$ for $i = 1, ..., k$, where $n = n_1$ and $n' = n_k$.

A hypergraph H is **acyclic** iff there is no path from any node in H to itself.

A set N' of nodes is **reachable** from a set R of nodes iff there is a path to each $n' \in N'$ from some $n \in R$. A hypergraph H is **reachable from** a set R of its nodes iff its set N of nodes is reachable from R. A **root** of H is a node from which H is reachable. A **leaf** of H is a node from which no other node is reachable.

If $H = (N, E, I, O)$ is a hypergraph, then its **opposite**, denoted H^{op}, is the hypergraph (N, E, O, I). We say that H is **coreachable** from N' iff H^{op} is reachable from N'. A **hyperpath** in a hypergraph $H = (N, E, I, O)$ **from** $D \subseteq N$ **to** $T \subseteq N$ is an acyclic hypergraph contained in H, reachable from D, and coreachable from T.

An **edge expansion** of a hypergraph is a replacement of a hyperedge by a hyperpath having the same input and output sets.

(We leave the notion of "replacement" informal here, because it is intuitively clear yet technically complex to write out in detail.)

We can now present the hypergraph version of the software evolution data model:

Definition: An **evolutionary hypergraph** is a hypergraph $H = (N, S, I, O)$ with a function $L: N \rightarrow C$ such the following assumptions are satisfied:

1. N and S are disjoint subsets of a set U whose elements are called *unique identifiers*;

2. if $O(s) \cap O(s') \neq \emptyset$ then $s = s'$; and

3. H is acyclic.

The elements of N are identifiers for software components, the elements of S are identifiers for evolution steps, I and O give the inputs and outputs of each evolution step, and the function L labels each node with a corresponding actual component from the set C of components. The notion of component used here includes both components in the usual sense and systems built by combining subcomponents.

The first condition says that the node and edge identifiers are distinct. The second says that the outputs of different evolution steps are distinct; this implies that each step is uniquely identifiable by the components that it produces. The third condition implies that the process of software evolution never brings us back to a component that we have already built; this simply means that we never reuse a unique identifier for a component. However, it is certainly possible that a later component is equal to an earlier one, in the sense that $L(n) = L(n')$ and n' is later than n, in a sense made precise by the following:

Definition: A node n' **depends on** a node n iff there is a path from n to n'. Similarly, a node n **depends on** a step s iff there is a path to n involving s. A step s' **depends on** a step s iff there is a path involving both s and s' with s earlier in the path than s'. We may say that a component c' **depends on** a component c iff there is a path from c to c' where $c' = L(n')$, $c = L(n)$, and n' depends on n.

The model so far developed does not include the idea that some evolution steps may be composites of other, lower level steps. To model this, we introduce a hierarchical structure on the hyperedges in a hypergraph. This also has the advantage of giving overviews of the evolution history at various levels of abstraction.

Definition: An (edge) **hierarchical hypergraph** is an acyclic graph with nodes labelled by hypergraphs, such that: the graph has just one leaf and one root; each of its edges corresponds to an expansion of a single hyperedge in its source hypergraph, the result of which is the hypergraph in its target; and the result of the composite expansions along any two paths between the same two nodes are equal. A **hierarchical evolutionary hypergraph** is a hierarchical hypergraph whose nodes are labelled by evolutionary hypergraphs.

The intuition behind this definition is that the root node hypergraph is the most abstract top level view of the system's history of evolution, while the leaf node is the fully expanded form of the system's evolution. All of the steps in the leaf hypergraph are atomic. The *part_of* relation in earlier versions of this model can be defined in terms of this graph of hypergraphs.

Work still in progress includes exploring node hierarchical hypergraphs and hypergraphs that are both node and edge hierarchical. Another issue not discussed here is imposing some additional structure on the set U of unique identifiers, in order to allow defining the notions of version and variant. This will result in further extensions to the model developed above.

2.3. An Evolution Model for Prototyping

Software evolution in CAPS can be described as a series of PSDL prototypes for each variation of the design. In simple cases, a prototyping effort can involve a single variation, i.e., a linear chain of versions, each derived from its predecessor, but in larger efforts several variations may be explored in parallel, by different teams, resulting in a more general acyclic dependency graph, as described in the previous section. The prototypes in a path of the dependency graph are iterative approximations $S_{k,i}$ to S_k (variation k of the software system S). Each prototype $S_{k,i}$ is modeled as a graph $G_{k,i} = (V_{k,i}, E_{k,i}, C_{k,i})$, where:

$V_{k,i}$ is a set of vertices, each of which can be an atomic operator or a composite operator modeled as another graph;

$E_{k,i}$ is a set of data streams, where each edge is labeled with the associated stream name; and

$C_{k,i}$ is a set of timing and control constraints imposed on the operators in version i of the prototype.

The change from the graph representing the i-th version in the k-th variation chain to the graph representing the $(i + 1)$-st version can be described by the following equations:

$S_{k,i+1} = S_{k,i} + \Delta S_{k,i}$
$\Delta S_{k,i} = (VA_{k,i}, VR_{k,i}, EA_{k,i}, ER_{k,i}, CA_{k,i}, CR_{k,i})$
$VA_{k,i} = V_{k,i+1} - V_{k,i}$: the vertices to be added to $S_{k,i}$.
$VR_{k,i} = V_{k,i} - V_{k,i+1}$: the vertices to be removed from $S_{k,i}$.
$EA_{k,i} = E_{k,i+1} - E_{k,i}$: the edges to be added to $S_{k,i}$.
$ER_{k,i} = E_{k,i} - E_{k,i+1}$: the edges to be removed from $S_{k,i}$.
$CA_{k,i} = C_{k,i+1} - C_{k,i}$: the timing and control constraints to be added to $S_{k,i}$.
$CR_{k,i} = C_{k,i} - C_{k,i+1}$: the timing and control constraints to be removed from $S_{k,i}$.

where the + operation above is defined by:

$V_{k,i+1} = (V_{k,i} \cup VA_{k,i}) - VR_{k,i}$
$E_{k,i+1} = (E_{k,i} \cup EA_{k,i}) - ER_{k,i}$
$C_{k,i+1} = (C_{k,i} \cup CA_{k,i}) - CR_{k,i}$

2.4. Change Merging

Automated support for combining several changes is useful in supporting fast prototype evolution, since responses to several change requests can be developed in parallel and then combined. If several different responses to an issue resulting from a prototype demonstration are developed, then a change merging facility can help to rapidly explore different combinations of several independent choices.

Two modifications A and C of the base version B of the semantic function computed by a software system can be merged according to the following formula [3]

$$M = A[B]C = (A - B) \cup (A \cap C) \cup (C - B),$$

where the union, intersection and difference operations are the usual set operations on relations. For example, the difference $(A - B)$ yields the part of the function present in the modification, but not in the base version. The intersection $(A \cap B)$ yields the part of the function preserved from the base version in both modifications. This model preserves all changes made to the base version, whether extensions or retractions, and two changes conflict if the construction produces a relation that is not a single valued function.

An approximate method for merging prototypes can be based on the change model and the above definition. Consider an i-th version which has been changed in two different ways, via Δ_k and Δ_l, corresponding to refinements of the variations k and l. The results of these two changes are denoted $S_{k,i+1}$ and $S_{l,i+1}$ respectively. Now let us consider a case where the $(i+2)$-nd iteration is the result of merging these two changes:

$$V_{k,i+2} = V_{k,i+1}[V_{k,i}]V_{l,i+1} = (V_{k,i+1} - V_{k,i}) \cup (V_{k,i+1} \cap V_{l,i+1}) \cup (V_{l,i+1} - V_{k,i})$$

The rules for combining the other components of S are similar.

This method is approximate in the sense that the change merging construction is applied to the structure of a PSDL program, rather than to the mathematical function it computes. The method is simple and efficient. Moreover, it corresponds to common programmer practice and produces semantically correct results most of the time; semantic correctness can be checked using the slicing invariance test explained below.

Prototype slicing is analogous to program slicing [3]. To do the slicing, we have to embed the PSDL graph in an augmented Prototype Dependence Graph (PDG). A PDG for a prototype P is a fully expanded PSDL implementation graph $G_P = (V, E, C)$ where the set of edges E has been augmented with a timer dependency edge from vertex v to v', when $v, v' \in V$ and v contains timer operations which affect the state of a PSDL timer read by v'.

A slice of a PSDL prototype P with respect to a set of streams X, $S_P(X) = (V, E, C)$, is a subgraph of the PDG G_P that includes the portion of P that affects the values written to data streams in X. A slice is constructed as follows:

(1) V is the smallest set that contains all vertices $v \in G_P$ that satisfy at least one of the following conditions:
   (a) v writes to one of the data streams in X;
   (b) v precedes v' in G_P and $v' \in V$.

(2) E is the smallest set that contains all data streams $x \in G_P$ that satisfy at least one of the following conditions:
   (a) $x \in X$;
   (b) x is directed to some $v \in V$.

(3) C consists of all of the timing and control constraints associated with each operator in V and each data stream in E.

The slice invariance theorem [4] says that the slice $S_P(X)$ of a prototype P with respect to a set of streams X has the same behavior as the entire prototype P on any subset of the streams in X. This theorem can be used to support a method analogous to the HPR algorithm [8] for combining changes to while-programs, and it can also be used to check the correctness of the approximate method as follows:

if

  the slice of the merged version with respect to the affected streams
  is the same as the corresponding slice of each modified version,

and

  the slice of the merged version with respect to the preserved streams
  is the same as the corresponding slice of the base version,

then

  semantic correctness of the merged version is established.

The slicing method has the advantage of providing a definite semantic criterion of correctness, and the disadvantage of reporting conflicts whenever two changes can affect the same output stream, regardless of whether there exists any computation history in which the two changes actually interact or conflict with each other. These advantages and disadvantages are analogous to those of the HPR algorithm [8]. The advantages of the approximate method are that it is simple and fast and can perform correct and successful merges in cases where the slicing method produces conflicts; its disadvantage is that it sometimes produces results that are not completely correct. Since failure cases for the approximate merging method are rare, it can be useful in the context of prototyping. We are working on improved methods for change merging that always produce correct results and report fewer false conflicts than the slicing method.

2.5. Retrieving Reusable Components

Reusable software is a promising approach to increasing software productivity, and seems especially promising in connection with the rapid prototyping approach to software development [16]. However, an effective way to retrieve reusable software components from a software base is needed for this approach to succeed: it is necessary to find the few components you want among the many you do not want. We call this the **search problem.**

It is important to notice that in practice, there may be *no* component in the software base that does *exactly* what is wanted, but there may be one or more components that can be easily modified to do the job. This implies that we should not look for a single exact match, but rather a set of approximate candidates. Thus any solution to the search problem should satisfy the following criteria:

(1) the search process should be fast, because a human user is waiting for the results;

(2) the choice set should not be too large; and

(3) the choice set should include the closest match, if there is one.

Luqi [9] has suggested associating a semantic specification with each module in the software base in order to support retrieval against semantic queries. This suggestion has been further developed in later publications [13, 16] using the OBJ3 [6] algebraic specification language to perform some experiments in the context of CAPS. Recent work [7] has carried this further by showing how to treat generic modules, how to use semantic information in a limited efficient way, how to achieve almost the same capability without associating specifications with modules, and how to rank candidate modules by their likelihood of success. Ranking modules by how well they satisfy the query yields a choice set of *ranked partial matches*. This can make the search process more robust, that is, better able to tolerate errors in the query and in the way components are classified. We should expect to encounter such errors in practice.

In this work, the components in the software base are assumed to be written in a modern programming language, e.g., Ada, that has strong typing, that can package together a number of operations over common data representations, and that allows generic modules having a number of parameter types and operations. Each component is assumed to have an executable algebraic specification with equations that are Church-Rosser. However, much of the work also applies if components do not have corresponding algebraic specifications. The user's query is also assumed to be a Church-Rosser algebraic specification. The components, their specifications, and the queries can all be *parameterized*, that is, the software base may contain generic modules, and queries may express the desire to find a generic module satisfying certain semantic properties.

Given a query Q and a component M having specification T_M, which is assumed to be a correct specification of M, a component M is a **correct answer** to the query Q iff there is a translation of the syntax of Q into the syntax of T_M such that all translated equational axioms of Q are consequences of T_M. These consequences may be either equational consequences or inductive consequences, depending of whether a "loose" or an "initial" semantics is given to the specification T_M. An important advantage of this approach is that it is insensitive to whether an initial or a loose semantics is assumed.

Finding a correct answer in this sense is a theorem proving task that could take so much time as to be impractical; however, finding candidates that satisfy adequate *necessary semantic conditions* for being a correct answer is a practical goal. This will allow many useless candidates to be rejected, thus narrowing the search dramatically and raising the confidence in the components found. After the appropriate components have been found, it may be worthwhile in some cases, e.g., safety critical applications, to fully verify that the components found are indeed correct.

We can organize the search as a series of increasingly demanding *filters* imposed on candidate components. The first step is to find components whose type structure is similar to that of the query. This form of type checking is accomplished by a process called *signature matching* which seeks maps that translate the type and function symbols of the query into corresponding type and function symbols of a candidate component. Matches of this kind can be *partial*, in the sense that only part of the functionality that the user seeks may be actually available. Once signature matches are found, more stringent semantic filters can be applied to further narrow the choice set. This can be accomplished through successive stages of *semantic validation* in which equations that are logical consequences of the query specification are translated by signature matches into equations whose proof is attempted in the candidate specifications. Even if specifications are not associated with components, we can still check whether ground equations are satisfied by applying the executable code to the terms on each side of the equation.

2.6. Real-Time Scheduling

Evolution of hard real-time systems in CAPS is supported by methods for automatically constructing real-time schedules and tools for automatically generating schedule code based on such methods [14]. This automation support is important in practice because small changes in the timing constraints can lead to very large changes in the programs that realize the constraints. Without automation support it is very difficult to modify real-time systems without damaging them.

The real-time schedulers in CAPS produce static and dynamic schedule programs to realize the real-time constraints given in the PSDL specification of the application problem. The schedulers check the feasibility of the timing constraints and produce schedules if feasibility is established. The static scheduler produces pre-run-time schedules for time-critical computations. The dynamic scheduler produces schedules for computations without hard deadlines. These computations use gaps in the execution of the static schedule, determined at run-time.

Automated scheduling depends on formal models of timing constraints. We have developed models of the timing constraints that occur in practice [15], taking care to make them as unrestrictive as possible, in order to admit as many feasible schedules as possible. Two categories of scheduling methods for a single processor are based on these models in the current implementation of CAPS:

(1) Heuristic methods, such as earliest-deadline-first, earliest-starting-time-first, and simulated annealing. These methods are efficient enough to handle large problems, but they can fail to find a schedule even if one exists.

(2) Complete search methods, such as branch-and-bound and exhaustive enumeration. These methods are guaranteed to find a schedule if one exists, but have exponential worst-case running times, and in practice can be successfully applied only to relatively small problems.

The currently implemented CAPS scheduling tools have been effective for the applications developed to date. Efforts to evaluate and improve them continue. The semantics of PSDL admits concurrent execution on an arbitrary number of processors, and extending the implementation to such architectures is a current focus of our research. We have developed some extensions to the above methods to handle multiple processors, and we are continuing efforts to characterize the effectiveness of these methods, develop improvements, and determine which methods are most effective on a large scale.

3. Conclusions

The goal of our research has been to help realize the potential benefits of formal methods for practical software development. Many researchers believe that this approach is the only hope for real progress in the long term, because automated decision support may be the only way to overcome the relatively high error rates characteristic of human decision making. Formal models and systematic methods for analyzing and manipulating them are needed to build tools that can go beyond fancy interfaces of manual design editors and provide effective tools for automating more of the decisions that software developers must make.

We have focused on software evolution because large-scale software development is dominated by constantly changing requirements and the difficulties of implementing the ensuing changes. Our experience indicates that formal models can provide useful decision support for software evolution. Much work remains to validate and improve the models, and to develop better computer aid for software evolution.

References

1. S. Badr, "A Model and Algorithms for a Software Evolution Control System", Ph.D. Thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, Dec. 1993.
2. V. Berzins and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1991.
3. V. Berzins, ed., *Software Merging and Slicing*, IEEE Computer Society Press Tutorial, to appear, 1995.
4. D. Dampier, "A Model for Merging Software Prototypes", Ph.D. Thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, June, 1994.
5. W. Gibbs, "Software's Chronic Crisis", *Scientific American*, Sep. 1994, 86-95.
6. J. Goguen, J. Meseguer, T. Winkler, K. Futatsugi, P. Lincoln and J. Jouannaud, "Introducing OBJ", SRI-CSL-88-8, Computer Science Lab, SRI International, Menlo Park, California, August 1988.
7. J. Goguen and J. Meseguer, *Retrieving Reusable Components*, SRI technical report, to appear, 1994.
8. S. Horwitz, J. Prins and T. Reps, "Integrating Non-Interfering Versions of Programs", *Trans. Prog. Lang and Systems 11*, 3 (July 1989), 345-387.
9. Luqi, *Normalized Specifications for Identifying Reusable Software*, Proc. of the ACM-IEEE 1987 Fall Joint Computer Conference, Dallas, Texas, October 1987.
10. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. on Software Eng. 14*, 10 (October, 1988), 1409-1423.
11. Luqi, "Software Evolution via Rapid Prototyping", *IEEE Computer 22*, 5 (May 1989), 13-25.
12. Luqi, "A Graph Model for Software Evolution", *IEEE Trans. on Software Eng. 16*, 8 (Aug. 1990), 917-927.
13. Luqi and Y. Lee, "Towards Automated Retrieval of Reusable Software Components", in *Proceedings of the AAAI Workshop on Artificial Intelligence and Automated Program Understanding*, San Jose, CA, July 13, 1992, 85-88.
14. Luqi, M. Shing and J. Brockett, "Real-Time Scheduling in System Prototyping", in *Proc. Fourth International Workshop on Rapid System Prototyping*, Research Triangle Park, NC, June 28-30, 1993, 150-163.
15. Luqi, "Real-Time Constraints in a Rapid Prototyping Language", *Journal of Computer Languages 18*, 2 (Spring 1993).
16. R. Steigerwald, Luqi and V. Berzins, "A Tool for Reusable Software Component Retrieval via Normalized Specifications", in *Proceedings of the Hawaii Conference on System Sciences*, Kauai, Hawaii, Jan. 1992, 18-26.

Whither Formal Methods?

Some thoughts on the Application of
Formal Methods
to the Problems of
Software Engineering

Daniel M. Berry
Faculty of Computer Science
Technion
Haifa 32000, Israel

Abstract

This note attempts to identify why formal methods are not being used in routine software development and uses personal observations to suggest that formal methods are not being used because they do not address the real problems faced by software developers.

1 Introduction

This note is written as a collection of personal musings (and if so-called facts turn out to be wrong, please pay more attention to the impressions reported herein; they are accurate) attempting to identify why formal methods are *not* widely, uniformly, and actively applied in industry to the development of production software. Previous attempts to identify reasons seem to focus on a lack of tools that assist in carrying out the tedious parts of formal methods, i.e., a technological problem. It is my belief that the problems are other than technological. They are simply put, that we formal methodologists are beating around the wrong bush; that is, we are *not* addressing the real problems faced by software developers in our research.

It appears to me that the main problem is that formal methods work has focused on what is possible rather than on what is needed. That is, researchers solve problems that are on the frontier between what is currently solvable and that which is not yet solved. They do not go out, find out what are the real problems and attempt to solve them, regardless of their difficulty or perceived lack of formalizability. While this is fine for basic theoretical foundational work, it is not acceptable for work purporting to be for the purpose of encouraging the use of formal methods to solve real software development problems.

Lest you believe that I attack formal methods from prejudice or a total lack of understanding, note that in my dark secret past, I used to consult for the formal methods group at Unisys Santa Monica, and at the time, I really believed in formal methods. The group made its money developing and selling a formal specification and verification method, the Formal Development Method (FDM), and associated tools. It also applied the FDM to specification and verification of security-relevant properties for a variety of customers, mostly branches of the government or military. One of my publications was a description of the semantics of the formal specification language developed there [Berry87]. I came to appreciate both the value and limitations of formal methods and their value in forcing redundancy. I learned the hopelessness of trying to prove a whole system totally correct; we focused on specifying security-relevant properties and proving only these of only the design and not the implementation. I also learned that formally specifying and verifying even only a small part of a full system drives the cost of the system up by an order of magnitude; consequently formal methods can be used only for very rich customers who perceive that the cost of a system malfunction is higher than this order-of-magnitude higher cost of development. In addition to this consulting position, of my 19 Ph.D. students, fully 7 of them worked on formal methods for software development; moreover these were 7 of my first 11 Ph.D. students.

2 Slow Progress

I used to be in programming languages and occasionally dabbled in the theory of programming language semantics. For that work, I had been acquainted with Hoare semantics, e.g. [Hoare71], and algebraic semantics, e.g. [Goguen79], and was aware of their limitations in dealing with pointers, which potentially could be aliases, and with permanently updated memory. The consequence of these limitations was that these formal semantics could not be the basis for verification of any real-live operating system in which pointers abound or of any real-live database system in which data are permanently modified.

As I gravitated towards hard-core software engineering, I lost touch with the semantics work. In 1992, I heard overview talks on these formal systems and other more recently developed formal systems. These talks described the same fundamental limitations as yet-to-be-solved problems, and expressed hopes for advances in the future. Furthermore, it appeared that the other formal systems had run into the same road blocks. I say this not to denigrate the people who do this work as not being smart enough to solve the limitations; it appears that the limitations are fundamental to the formal models used.

The reality is that the problem is a tradeoff between power of expression and simplicity of the resulting formulae. One can always model memory in a more complete formal model than the one typically used, in which the mapping from identifier to value is direct. To handle pointers and memory, it is necessary to make pointers be first class values and map from identifiers to pointers and from pointers to values in two separate mappings. However, the cost is a dramatic increase in the size and complexity of formulae with which one describes the software. Basically, the memory has become an explicit parameter of all functions. A formal system whose formulae look like programming language statements, in which memory is an implied, hidden parameter, is inherently limited, *unless* some way can be found to hide the memory in the formal system.

Indeed, Joseph Goguen [Goguen94] reports precisely such a solution in which he distinguishes between visible and hidden sorts, with the latter representing states. He lets equations be satisfied only *behaviorally*, and he uses loose semantics, so that any representation of states is allowed. With the earlier initial algebra semantics approach, an explicit representation of states would need to be given, but with this new approach, no representation at all is needed, and that gives a tremendous simplification. The approach fits the object paradigm very closely and views ordinary imperative programming as a special case of that paradigm, in which the cells that hold the values of variables are the objects. One objective has been to make proofs as simple as possible, with just reduction and a bit of induction and *no* messing with quantifiers and the like. He reports that the current work is on concurrency. The emphasis so far has been on foundations, language design, and tools, and he hopes to do some larger examples soon.

Clearly progress has been made, but relative to the kinds of programs people are writing, progress has been slow. That is, while we have gone through three generations of programming languages since the foundations of Hoare semantics were introduced, we are still struggling to be able to prove statements about the bodies of procedures that we could write back then.

I picked the pointer and memory examples to demonstrate slow progress merely because the limitations are precisely what has kept Hoare logic and algebraic semantics from being fully applicable to the kinds of ordinary, nonconcurrent software and system programs that make up the bulk of software that has been and still is being written.

This is not to ignore other areas of formal methods progress. For example, state machine models have been applied with great success in modeling protocols [Sunshine82]. These formal models have been instrumental in the design and implementation of demonstrably correct and stable network protocols. Indeed, it is said that formal protocol models have exposed anomalies that have actually occurred, bringing the ARPA Net to a screeching, grinding halt.

3 Suggesting Use of Limited Languages that Permit Verification

I used to snicker at the formal methodologist that beseeched programmers to write purely functional programs with no side effects, no pointers, and no permanent update to memory in order to make it possible to prove programs correct. It was really pathetic. It appeared to me that these people had never written a large software system and certainly none of their published examples ever got beyond the toy program scale. Some of them admitted the limitations of such languages, but countered with "But your software will be verifiable!" as if those who would not use functional languages because of their limitations really cared.

Moreover, I as a programmer use pointers and gotos with impunity. As a programmer, I have never understood why they are so nasty. They certainly have never given me as much difficulty as one might expect from the vast literature explaining why they should be considered harmful. I have an accurate mental model of their behavior [Johnston71], and my software using them never seems to suffer from spaghetti-itis. It seems in retrospect that the strongest claims of nastiness come from those who have seen their use as an impediment to formal verification.

Indeed, in a report that I recently wrote for the SEI [Berry92], I made the following observation about the responsibility of researchers claiming to have a technique for improving practical software development. "When any approach is suggested by researchers, those researchers must, as part of their job, assess the effectiveness of the ideas and then determine if that assessment yields a statistically significant demonstration of the effectiveness of the approach. This evaluation is necessary, regardless of the nature of the approach, be it foundational theory, a tool, an environment, a method, or a technique. Even when the approach is theoretical and the theory can be proved sound, the researcher must demonstrate the relevance and usefulness of the theory and the effectiveness of its application to the production of reliable software.

Too often, software engineering researchers propose an idea, apply it to a few, toy-sized examples, and then grandiosely jump across a gap as wide as the Grand Canyon to an unwarranted conclusion that the idea will work in every case and that it may even be the best approach to the problem it purports to solve. Some of these researchers then embark on a new career to evangelically market the approach with the fervor of a religious leader beckoning all who listen to be believers.... While historically, methodologists have the biggest reputations for this behavior, they certainly have no monopoly on it. Theoreticians are equally guilty of proposing a formal method of software development, showing that it works for a small, previously solved, formally definable problem, and then expecting the rest of the world to see that it will work for all problems. When the rest of the world does not see how to apply it to large, real-world problems and wonders why the theoretician does not go ahead and apply it to such problems, the theoretician throws his or her hands up in the air in fit of despair over the sloppy way programmers work."

4 Little Believed Benefits of Formal Methods

I also used to snicker at papers, books, and presentations by formal methodologists in which the toy-sized example demonstrating a formal method had errors, bugs, just like ordinary programs that I and others write. If it were a live presentation and someone took the speaker to task, he or she might say, "Oh well! In any case, you see what I mean?" and would still claim at the end that his or her formal method would help eliminate or at least reduce the occurrence of errors. Nonsense! Yes, I can see what you mean. You make as many errors as the rest of us do without formal methods!

Having written numerous formal specifications, I would say that programming and formal specification are the same. Both require a complete formal model of a real-world situation, both require much revision as they are being written, both are equally prone to errors, and both require debugging of errors found in an attempt to run or verify them. Goguen adds parenthetically, as he is indicating that an example specification written in an executable specification language has in fact been executed [Goguen93], "(Experience shows that it is necessary to test all but the most trivial specifications in order to eliminate bugs.)" Sometimes it is hard to see the benefit of applying formal methods.

5 Would a Formal Methodologist use Formal Methods to Develop Mathematical Theory?

Indeed the whole idea of using formal methods is that one writes two formal descriptions of the same system and proves them equivalent. Failure to be able to carry out the proof exposes errors, and correction of these errors often requires changes to both formal descriptions. Programmers are being asked to double their work by producing two formal models of the same real-world system, i.e., the program and say, input-output assertions, and they are being asked to do that about something that is not fully understood, i.e., the customer's requirements. To understand how irrational this expectation sounds to programmers, consider the following. We all know how error-ridden are mathematical papers, which purport to prove many theorems. Indeed, a number of published so-called proofs are not correct, even though the theorem they claim to prove might in fact be theorems. A smaller number of published so-called theorems are not even theorems. Perhaps we should insist that mathematicians work with formal methods the way we expect programmers to. All formal systems shall be developed in two different notations and the two versions shall be proved equivalent before a paper about one of them can be published. Perhaps we should insist that mathematicians document the development of the formal system and show in this documentation that the formal system was developed in a systematic formal way. This is sheer nonsense, and yet we expect this of programmers.

6 Formal Methods in Formal Method Tool Development

As I mentioned, I used to consult in the formal methods group at Unisys. We were developing tools including a specification language compiler to verification conditions and a theorem prover. Initially, these had a line-by-line, interactive, ASCII character user interface, and later we began to develop a more user-friendly, windows-based, graphic user interface. The big question that almost no one asked, perhaps out of embarrassment or fear of the answer, was "Did we use formal methods to develop these tools, and if not, why not?". We did not, and the reasons we would have given are enlightening:

1. It was not really necessary to specify and verify the software because most of it was input-output anyway and did not involve algorithms worth verifying the correctness of their implementation.

2. It was not really necessary to specify and verify the software because most of it was organizational anyway and did not involve any algorithms worth verifying the correctness of their implementation.

3. We wrote good and careful code anyway.

4. Because the theorem prover is not a function (when implemented correctly, it is nonterminating), it is not possible to specify input-output behavior.

5. It is not possible to specify and verify the software because the hard parts deal with the user interface.

6. The code that we would have to verify was too big to be verified anyway.

7. We did not have the time to do the verification, as we were under a tight schedule.

8. There were no tools available to help us (mainly because we were developing them).

These are the same reasons we hear from experienced programmers as to why they do not practice formal methods. I wonder how many formal methods tool developers use formal methods to develop their tools?

7 An Ethic

In another of my research areas, electronic publishing, there is an ethic that is followed by authors of papers about formatting software and developers of formatting software. The paper and the user's manual must be formatted and typeset using the software itself. If the paper is for a journal, then the paper must be formatted and typeset camera-ready in the journal's standard style so that it is not possible for even the expert reader to spot that the author and not the journal publisher typeset the paper. I have proudly followed this ethic with all such papers published in the journals *Software—Practice and Experience* and *Electronic Publishing*, the latter of which supports the ethic, and in the Electronic Publishing Conference proceedings [Abe89, Becker90, Habusha90, Wolfman90, Srouji92]. Moreover, I have been known to take a famous formatter author to task for not forcing his journal editors to let him typeset camera-ready with his own software a seminal article about some lessons learned while developing the software. Formatting and typesetting a user's manual about a formatting program with the program itself is a great test of the software and of the usability of the software, especially if all the features of the software are described by examples in the manual. Developers of tools for formal methods should feel compelled to apply the methods and tools themselves (in later stages) to the development of the tools themselves and their later enhancements.

8 An Analogy between Programming and Building a Theory

More difficult than building a formal system is deciding what should be in the formal system. I can recall building a formal system from ground up. I had to change it several times before I got it right. It took several tries to get the definitions and axioms to say exactly what they should so that I could get the desired theorems rolling out. I found that getting the foundations right took far more effort and creativity than doing any of the proofs involved. This is exactly the situation in software development. Figuring out what should be in a system is much harder than building the system once it is figured out what should be in it. It is often necessary to build versions of the system (prototypes) in order to see what should be in it, just as it was often necessary to go through a whole series of definitions, axioms, and theorems to see that a different set of definitions and axioms are needed to get the desired theorems.

No more than one can consider using a formal method to decide what should be in a formal system can one consider using a formal method to decide what should be in software. These are a matter of taking what is in someone's or some people's heads and in their environment and expressing it in language. The first expression is more difficult to achieve than a formal expression once a first expression has been achieved. Section 13 on requirements engineering offers some clues as to why it is so difficult to decide what should be in a system.

9 Failed Attempt at University-Industry Cooperation

About a year ago a group of Technion Faculty of Computer Science software engineering faculty members went to talk with the chief software engineer at a local high-technology company doing major software development. The chief software engineer explained the problems that the company had. These were primarily in gathering complete, consistent, and useful requirements from clients, change management, and configuration management, none of which have total suitable technical solutions. For the first, the problem is getting information from the insides of the brains of different people, information that may not even be there as the person has not even thought of everything. For the second, the problem is maintaining consistency between related parts of a system and its documentation in the face of changes. For the third, the problem is keeping track of different revisions and versions of the same or related programs and of keeping identical or related those parts that should be identical or related in the face of modifications.

The first has no solutions to date, and the other two have solutions that depend on programmers remembering to look for and update related parts and to update them consistently, on remembering to check a module out for modification and check it back in when finished, and on remembering to write useful documentation. If the programmers forget any of the required steps, the technical solution ceases to be effective (more on this subject later).

One of the visiting faculty members, whose name will be kept anonymous to protect the guilty, said that these problems were "not scientific enough" (whatever that means?) and described various projects on which he was working in methods for object oriented programming, tools for managing inheritance hierarchies, and semantics of inheritance. Of course, the chief software engineer had no interest in these topics. I personally could not contribute any research ideas at this meeting because I was going on leave just as the joint work would be starting; so I sat and just listened. I am quite sure that the chief engineer has no respect for our department's software engineering group.

10 Some Bad Formal Methods Papers for Software Engineering Conferences

I have been refereeing papers for various software engineering conferences for quite some time now. There is a particular kind of paper that I almost always reject. I describe a particular paper, again keeping the name of the author anonymous in order to protect the guilty. The paper is about choosing a formal method to use for writing system requirements.

It says that formal methods have "often" been used in the requirements specification phase. This is total nonsense; in my most charitable mood I might concede to the use of the word "occasionally".

The paper also claims that there is "some evidence" that precision and reliability "can be improved" by including formal specification in requirements specification and design methods because of its mathematical and logical principles and the fact that it avoids ambiguity. My first remark was that there is *some* evidence but not much. However if all that is required of the evidence is to show that precision and reliability *can* be improved, then I have to yield. Giving milk and cookies to programmers every morning before they start to work *can* help them improve the specifications they write. The real question is "Is there evidence that use of formal methods *improves* the precision and reliability of requirements specifications"? As far as I know, there is no such evidence. It is an accepted tenet among formal methodologists that use of formal methods does improve the precision and reliability of requirements specifications (and from the point of view of a formal methodologist actually doing the work, this belief is enough—the perception guarantees the reality). However, I have not seen any experimental verification of this claim. If there are any, I would be pleased to hear about it and to be shown wrong. In my own experience, the hard problem in requirements engineering is figuring out *what* the requirements should say and compared to this problem, that of obtaining formal requirements specifications from an understanding of the requirements is relatively straightforward. There is more on this issue below.

This same paper attempts to identify the reasons that formal methods are not used in software development. It offers the following:

1. Most systems analysts do not have the skills to develop formal specifications of systems. In addition, the skill of identifying abstractions (what to formalize) is not yet something that formal methods experts know how to teach.

2. Some classes of systems are difficult to specify using the present specification techniques; there is no formal method that adequately expresses nonfunctional properties such as performance, reliability, and quality of user interface, etc.

3. There is a lot of ignorance of the existence and applicability of formal methods.

4. More effort must be employed to develop support tools that are essential for dealing with specifications of large systems. Current specification and verification environments have not been used on large systems yet [This is not quite true, but the perception is important].

The first and third of these require education of programmers to be different from the start; there is no real reason why it cannot have begun a long time ago. Indeed, we have been making the same claim about the need for educating programmers in this way for decades; presumably those of us in academia *have been* educating programmers this way; and the fact that we still press for the same changes in programmer education says that this education has failed to have the desired effect. The second exposes some fundamental research problems for formal methodologists. There appears to be relatively little work in these problems, and I do not understand why. It is however, clear that these problems are tough and therein may lie the reason that not many are working on them. The preference is to formalize what is well understood and these properties are just not well enough understood to be formalized. The fourth may not be all that useful. That is, people have been arguing for decades that software engineering needs tools and environments, and yet the tools and environments that are supposed to help are not used as much as one might expect. Perhaps, the conclusion of those who have used them is that their benefits do not warrant their cost or the time spent to learn them.

11 Some More Bad Formal Methods Papers About the Need for Better Notation

I have read many a paper which starts off by lamenting that formal methods are not applied to routine software development. It observes that one reason is that the current methods are not programmer-friendly and shows how difficult writing specifications or verification can be using currently available languages and systems. Then it gives a *new* notation, language, method, system, etc. together with a few toy examples of its use. Then the paper ends. I have yet to see a paper following this model evaluate its own notation, language, method, or system against the criticism rightfully leveled against others. It is quite clear that the new notation, language, method, or system is perfectly clear and easy to use for the author, but I personally find all such new notations, language, methods, and systems to be as nonperspicuous as all the others and that I could level the author's criticisms against his or her own contribution. The conclusion I come to as a referee is that the author is interested in presenting his or her own system and is not really interested in how useful it is; after all, a formal system does not have to be useful to be beautiful. In a sense, the paper would be better if the contents were presented strictly as a new formal system presented on its own merits, instead of trying to dress up the work as being useful and therefore acceptable to a software engineering conference or journal.

12 Theory vs. Practice in the Eyes of Practitioners

The perception of practitioners is that theory lags behind practice, i.e., that practice has solved more problems and gotten more systems built than has theory. Robert Glass lists topics in which software practice leads software theory [Glass94]. These include software design, software maintenance, programming-in-the-large, software modeling and simulation, and software metrics. He observes that in one area, specifically user interface design, theory *has* significant contributions to the point that it can be considered as running neck and neck with practice.

Perhaps practitioners are telling us that we are solving the wrong problems, ones that they have solved themselves years ago. Perhaps it would be better for us to put our abilities to use to solve the problems that practitioners have not solved.

It is interesting to compare Glass's list with the list of topics on which the company software engineering department head wanted work at the university-industry meeting described above.

13 Requirements Engineering

One of my main areas of research these days is requirements engineering. In this section, I attempt to describe what I believe are the problems and to indicate why I do not see much help from formal methods.

In January 1994, I prepared a lecture titled "The Requirements Iceberg and Various Icepicks Picking at It", which showed what I believed were the problems facing people who have to write software requirements. There were two pictures in this talk that convey more than two thousand words worth of why I believe that requirements engineering is so difficult. The first, shown as Figure 1, shows that what a given client representative can tell you about the system the client wants is but a tiny fraction of what will have to be specified to have complete requirements. Note also the size of the instrument being used by the unhappy requirements analyst. The second, shown as Figure 2, shows that I believe that determining what to specify is far more difficult and tortuous than formalizing it once it has been determined. I do recognize that it is often useful to use the formalization process as the driver of questions that are posed to the client.

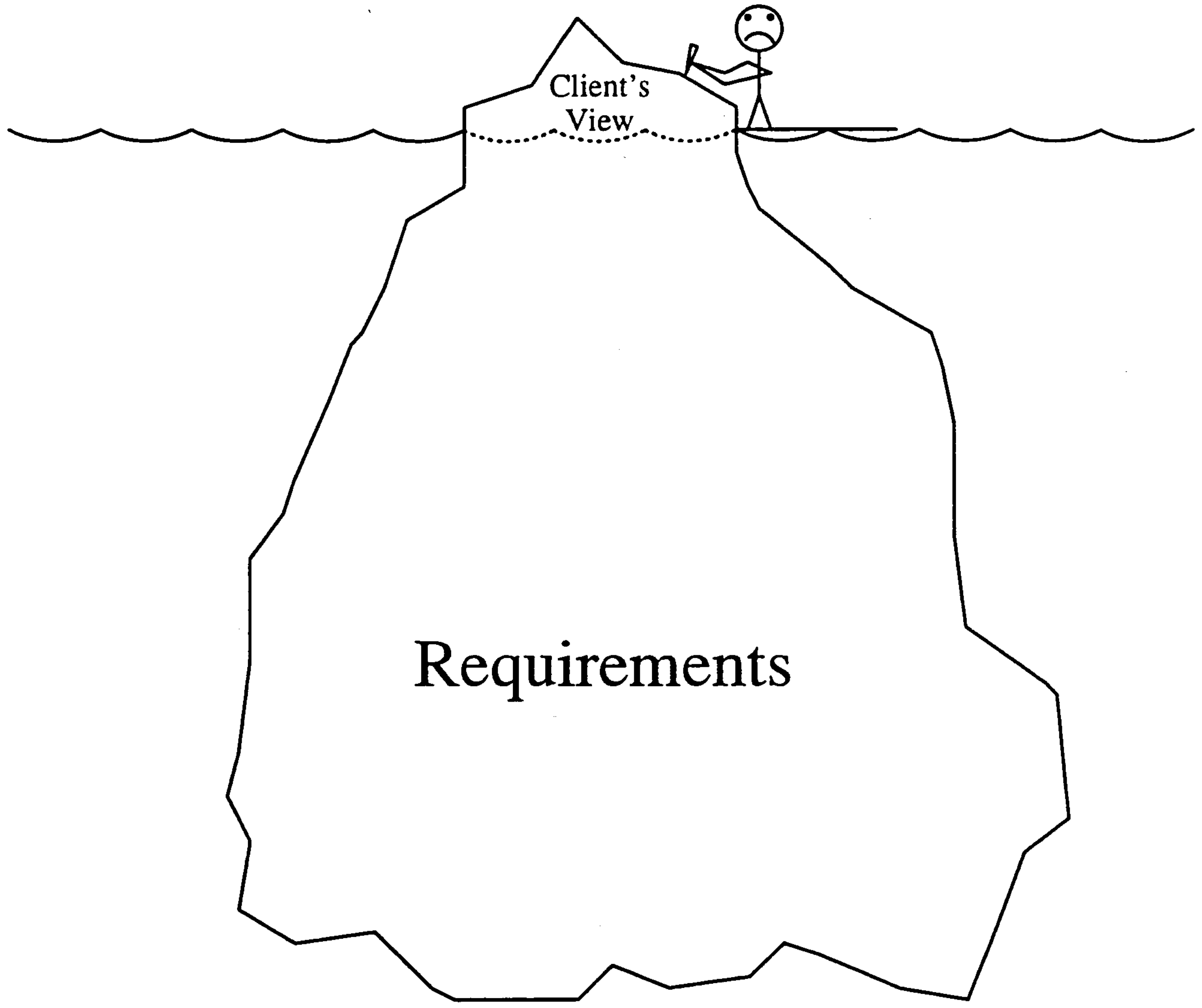


Figure 1: The Requirements Iceberg

Meir Lehman identifies E-type software as a software-based system that solves a problem or implements an application in some *real-world* domain [Lehman80]. Once installed, an E-type system becomes inextricably part of the application domain, so that it ends up altering its own requirements. Such systems are extremely impossible to understand and specify fully.

Johnny Martin and W. T. Tsai report on an experiment to identify lifecycle stages in which requirement errors are found [Martin88]. A user had produced a *polished* 10-page requirements document for a centralized railroad traffic controller. Martin and Tsai arranged for ten 4-person teams of software engineers to search for errors in this requirements document. The user believed that only one or two errors would be found. Instead 92 errors, some

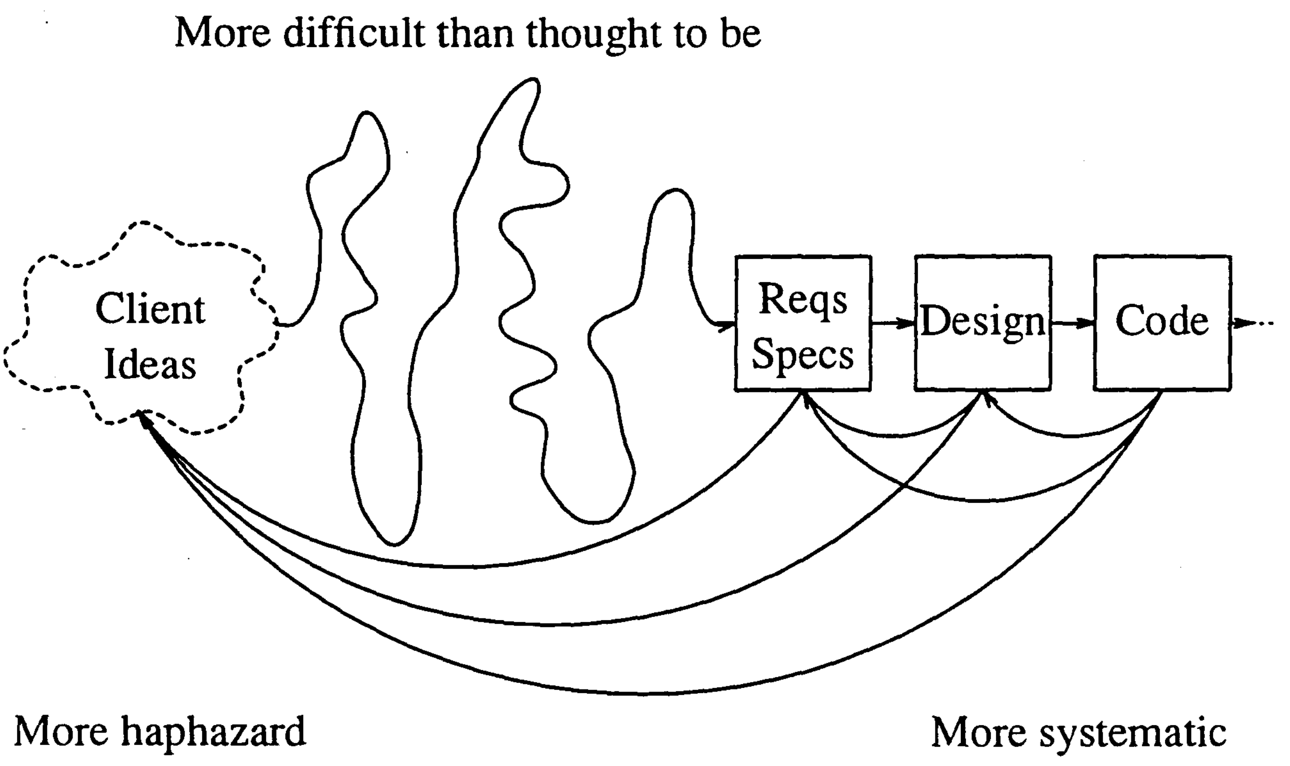


Figure 2: The True Software Life Cycle

very serious, were found. Worse than that, the average team found only 36.5 of the 92, leaving 56.5 to be found later downstream, where it is more expensive to do so. Finally, the most serious errors were found by the fewest teams. It is not clear how much formal methods would have helped. My own feeling is that many of the errors of total omission would have been missed in a formal specification as well, but it would be worth doing an experiment to this effect. This might be a good experiment as to the effectiveness of formal methods.

Goguen [Goguen94] has observed that it is not quite accurate to say that requirements are in the minds of clients; it would be more accurate to say that they are in the social system of the client organization. They have to be invented, not captured or elicited, and that invention has to be a cooperative venture involving the client, the users, and the developers. The difficulties are mainly social, political, and cultural, and not technical. Thus, social science methods are needed.

Unfortunately, these social science methods, based on an orthodox science approach of formulating a theory and then testing empirically the holding of predictions derived from the theory, suffer from the fact that the very observation of data affects the data in ways that can invalidate the conclusions. This effect is particularly acute in testing hypotheses about social situations because the observation becomes part of the social situation. It is necessary to accept that observation of social situations can affect the results. Goguen offers ethnomethodology as a general class of methods that accepts this fact and focuses on making observations that hold up even under the fact of being observed [Goguen93].

For example, the traditional approach of the requirements engineer's interviewing members of the client's organization does not expose what people really do, because people cannot describe what they really do very well, and direct questioning does not ferret out tacit assumptions very well, because the questioner does not know what to ask. Instead, Karen Holtzblatt suggests contextual inquiry in which the requirement engineer becomes a functioning member of the client's organization long enough to blend in and to observe him or herself what really happens and to learn about unspoken assumptions the same way that any new employee learns the ropes.

Goguen also observes that most of the effort for a typical large system goes into maintenance. (The actual data reported by David Parnas [Parnas94] are captured by the graph shown in Figure 3.) This, some formal methodologists say, is the fault of insufficient effort put into being precise in the early, specification stages of software development. However, Goguen believes that "a deeper reason is that much more is going on during so-called maintenance than is generally realized. In particular, reassessment and re-doing of requirements, specification, and code, as well as documentation and validation, are very much part of maintenance...." Later, he adds, "it only becomes clear what the requirements really are when the system is successfully operating in its social and organisational context.... it is impossible to completely formalise requirements ... because they cannot be fully separated from their social context." Goguen, in essence, describes precisely the phenomenon of E-type systems.

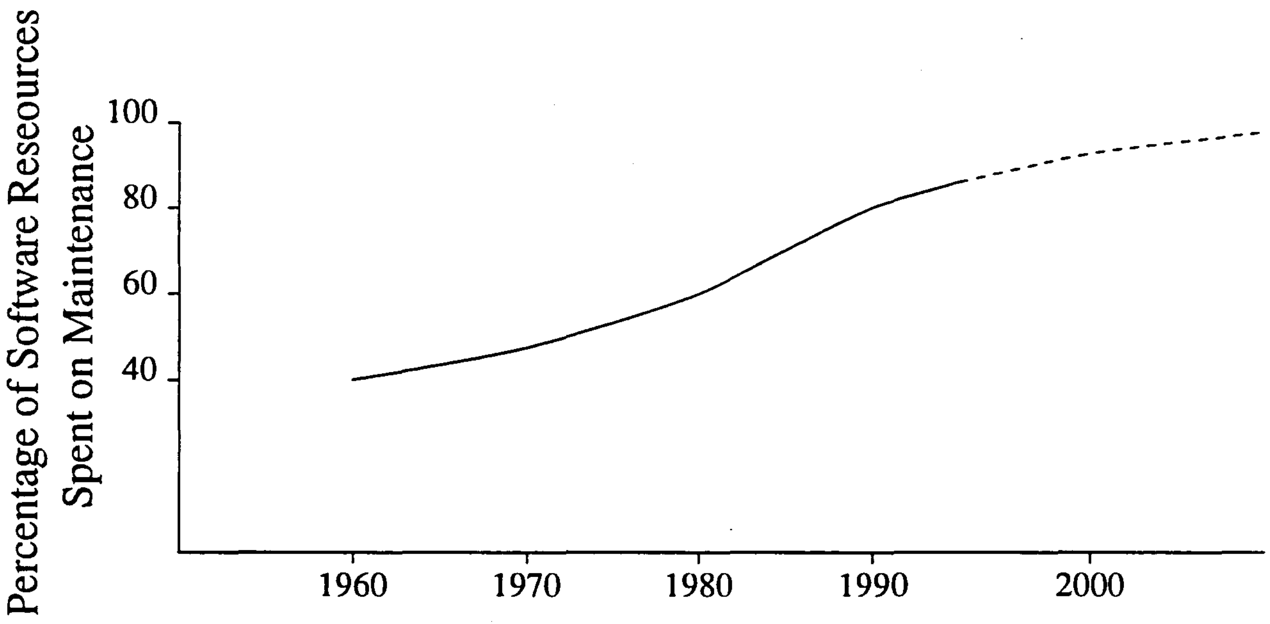


Figure 3: Growing Percentage of Maintenance Costs

What does the requirements engineering community think are the problems in requirements engineering? Recently, I received a call for papers for the Second IEEE International Symposium on Requirements Engineering (RE '95). It asked each submitter to classify his or her paper according to a classification scheme that can be obtained by anonymous FTP from the program chair. As I was submitting a paper, I obtained the scheme. It is instructive to reproduce that scheme here, because it exhibits what some program committee members consider to be the problems of requirements engineering:

Second IEEE International Symposium on Requirements Engineering
OFFICIAL SYMPOSIUM CLASSIFICATION SCHEME FOR SUBMITTED PAPERS

PROBLEMS OF REQUIREMENTS ENGINEERING

Research in requirements engineering can be classified according to the problem(s) being attacked. Note that a problem is different from a requirements-engineering task such as elicitation or specification, because not all tasks are problematic, and some problems affect many tasks.

1. Problems of investigating the goals, functions, and constraints of a software system
   - 1.1. Identifying client groups and interests
   - 1.2. Overcoming barriers to communication
   - 1.3. Converting vague goals (e.g., "user-friendliness," "security") into specific properties
   - 1.4. Allocating requirements to the software component of a broader system
   - 1.5. Understanding priorities and ranges of satisfaction
   - 1.6. Estimating costs, risks, and schedules
   - 1.7. Ensuring completeness
2. Problems of translating goals, functions, and constraints into specifications of software system behavior
   - 2.1. Generating and evaluating alternative strategies for meeting requirements
   - 2.2. Integrating multiple views and representations
   - 2.3. Engineering trade-offs and optimizations
   - 2.4. Obtaining complete, consistent, and comprehensible specifications
   - 2.5. Checking that the specified system will satisfy its goals, functional needs, and constraints
   - 2.6. Obtaining specifications that are well-suited for design and implementation activities
3. Problems of managing evolution
   - 3.1. Ensuring that the results of requirements engineering are modifiable and maintainable
   - 3.2. Extending or improving ill-structured systems
   - 3.3. Identifying and exploiting the common characteristics of a family of systems
   - 3.4. Reusing the artifacts of requirements engineering
4. Other (please specify)

CONTRIBUTIONS TO SOLUTIONS IN REQUIREMENTS ENGINEERING

Research in requirements engineering can also be classified according to its contribution(s) to a solution. Note the implicit assumption that, as software engineers, we can seek to understand social factors but we can only hope to influence technical practices.

A. Report on the state of practice
B. Analysis of cultural, political, organizational, and economic factors relevant to a problem
C. Proposed process-oriented solution (an orderly method for making decisions or accomplishing a task)
D. Proposed product-oriented solution (focusing on the representations used and produced by tasks, and algorithmic manipulations of these representations)
E. Case study applying a proposed solution to a substantial example (for the purpose of gaining experience and preparing for a more systematic evaluation)
F. Evaluation or comparison of proposed solutions
G. Other (please specify)

Of the 17 lowest level problems, I personally classify only 5 of them (1.4, 1.6, 1.7, 2.3, and 2.4) as being technical; the rest are decidedly nontechnical and go into human behavioral issues and how to get thoughts out of the minds of the client representatives. I classify the topic of the paper that I was submitting similarly as nontechnical, as it goes into the human use of language. I classify none of the contributions as being technical. If formal methods are to have a significant impact on requirements engineering beyond being used to write the requirements, formal methods will have to come to grips with the 12 nontechnical problems as well.

I might add that I found this list to be incomplete, because I could not find a suitable problem classification for the subject of the paper I was submitting about scenarios for abstraction identification in the natural language text given to the requirements engineer by the members of the client organization. It appears to be incomplete in both the technical and nontechnical areas.

14 Software Evolution

The theme of this workshop is "Software Evolution". The industrial chief of software engineering that I described earlier would agree that software evolution is a critical problem, as she asked us to work on two of its subproblems, change management and configuration management. Specifically the problem in her company was to keep track of relations between modules of programs as they evolve. These evolutions include correcting and enhancing programs as well as adapting them to different hardware and operating system platforms.

Luqi, in an effort to formalize parts of the problem and to identify candidate parts for formal method assistance, has written a graph model for software evolution [Luqi90]. Sofware evolution involves change requests issued by members of the organization of the customer of a computing system. These request must be reviewed by the management of the development organization and if accepted, must be scheduled for implementation by the software engineers.

A key problem is that even small changes can affect functionality and performance in major unforeseen ways. Part of management's job is to determine the effects of a change request and to judge whether to allow the change on the basis of these effects. In general, the only way to prevent adverse effects of changes is to be fully aware of *all* dependencies between all parts of the software. The number of such dependencies grows exponentially with the size of the software, no matter how the size is measured. The problem quickly outstrips our ability to manage, *even* with formal methods. Moreover the determination of the effect of any change on any known dependent part is not computable. Luqi points out the "an important practical problem in the evolution of a large system is ensuring the consistency of each new configuration. While the certification of semantic consistency involves several computationally undecidable problems in the general case, some related consistency criteria based on structural considerations can be maintained automatically with practical amounts of computation." She proposes the induced evolution step as the embodiment of such automatic maintenance.

My own personal experience using a tool that exposes parts of a program that are certainly or possibly affected by a change is that the tool helps with the easy part, for which I do not need help, and leaves the hard part, for which I do need help, to me. The tool helps find parts related by variables but has difficulty finding parts related by computed pointers. (Maybe that's what I get for using pointers!) The tool does not give much help in determining the impact of a change on a related part, especially when I cannot see that impact myself. Much work needs to be done.

Finally, a basic flaw in many a configuration management tool is that unless *all* components and documents of the emerging system are controlled by the tool and unless the tool is used consistently and religiously from the beginning with no lapse, the missing information can kill the tool's effectiveness.

The situation for rcs [Tichy85] is typical. It is built on top of the basic file system and requires programmers to remember to check out modules to be modified and to check in modules that have been modified. It requires programmers to document changes in modules as they are being checked in. If everyone has properly checked modules out and in, then rcs tells programmers who have modified the same module concurrently to sit and resolve the modifications. However, it is too easy to forget to check a module out or in. It is too easy to enter nonsense when being forced to enter a description of the changes as a module is being checked in. Finally, rcs has no way to verify the claim that the authors of concurrent modifications to the same module have actually met and written a consistent new module. So, here too, the problem is human behavior and not technical.

15 What I had Proposed for the Workshop

After being invited to the present workshop, I began to think about the problems to be addressed by the workshop and what I might write for this present paper. I thought of most of what was written above. Then, it occurred to me that perhaps a different organization might be useful for the workshop. Below is a copy of nonpersonal parts of a letter that I sent to Prof. Luqi in an attempt to change the direction of the present workshop.

7 June 1994

Dear Luqi;

I have looked at the description of the workshop and have some ideas how it can be made even more effective.

To quote the description of the problem,

"DoD and the computer industry urgently need software systems that can meet user needs effectively and reliably. Formal methods that can be partially or completely automated provide a fundamental approach to this problem, because they influence the design of languages for programming-in-the-large and because of their application to specification and analysis tools. To enhance software quality, formal methods should play a more prominent role in the software production cycle. However, most developers are not using formal methods, and researchers have not addressed many of the issues that arise in large scale applications of formal methods.

To remedy this situation, researchers in formal methods need to better understand where the developers of large software systems need help. Also software developers need to understand the benefits of formal methods, and of software tools that apply these methods to practical problems. The workshop will assess the practical impact of formal methods and tools, identify gaps between the capabilities of formal methods and the practical needs of software development, and define appropriate research directions. The workshop will also provide an opportunity to share recent advances in formal methods and their integration in software development environments."

It is observed that, "Formal methods and tools do not yet adequately support the development of large and complex systems. Many research efforts on formal methods have focused on narrow parts of the problem, and are difficult to integrate. We have no accurate models of the software development process and its relation to internal software structures. Existing models have either focused on narrow aspects of the process or have tried to cover the entire software life cycle through informal approaches that cannot support automation."

To help remedy the problem, "Presentations given by speakers who are actively involved in developing formal techniques and tools for different aspects of computer aided software development will be interleaved with periods for related discussion. Potential topics include the following:

[I]. Review the state of the art in formal methods related to the workshop focus and their use: What is currently feasible, and what is likely to be feasible in the near future without any major breakthroughs? ...

[II]. Review the problems and barriers faced by software developers in using and integrating formal methods in their work: How can formal methods help in the software development cycle and what are the major problems that need to be solved?"

I fear that the format implied by the above description will not be as effective as it could be.

First, let me say that the goal of the workshop, that of identifying ways that formalisms can help industrial strength software development, is critical. I also note that you have assembled quite a group of leaders in formalisms in software engineering. I am flattered to be considered among them.

I would like to propose that a major focus of the workshop should not be for we researchers to describe what we think should be done and what we can do. Rather it should be for we researchers to hear what industrial software engineers say are the problems. Perhaps we should spend two of the three days listening to a series of invited industry people describe their software development problems, what *they* would like to see solved, and what *they* believe has a chance of being solved. Then perhaps on the third day, we researchers could brainstorm about directions based on what we have learned.

I could propose a few industrial people, some of whom have had extensive academic experience (e.g., have Ph.D.s or teach in Universities). We could even stick to people from the Bay Area to save money and given the industry available, that would not be a severe limitation. I am thinking of people such as [list of specific names omitted]. Certainly, others among the invitees' list can propose other, even, better people.

No matter what is decided, I am prepared to participate fully and enthusiastically.

Sincerely,
Dan

Evidently it is too late to change the structure of the present workshop, and this is to be expected, given the large amount of work that had already gone into preparing for this workship. I offer this letter as an idea to put into effect at the next instance of this workshop.

16 Conclusions

This note offers glimpses at what I believe are reasons that formal methods are not being used in the development of production software. It proposes a modification to the structure of this workshop that will give us formal methodologists information with which to alter our research directions.

I strongly believe that people who work in formal methods have strong, disciplined minds that are capable of deep, creative thought. It is necessary for these people to use this mental power to solve the problems that are there and not the problems that are generated when considering formal methods.

Acknowledgments

I thank Jim Brockett, Joe Goguen, and Luqi for their comments on a previous draft of this paper. These led to what I believe is a better paper.

Bibliography

[Abe89] K.K. Abe and D.M. Berry, "indx and findphrases, A System for Generating Indexes for Ditroff Documents," *Software—Practice and Experience* 19(1), p.1–34 (1989).

[Becker90] Z. Becker and D.M. Berry, "triroff, an Adaptation of the Device-Independent troff for Formatting Tri-Directional Text," *Electronic Publishing* 2(3), p.119–142 (1990).

[Berry87] D.M. Berry, "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).

[Berry92] D.M. Berry, "Academic Legitimacy of the Software Engineering Discipline," Technical Report, CMU/SEI-92-TR-34, Software Engineering Institute (1992).

[Glass94] R.L. Glass, "A Tabulation of Topics where Software Practice Leads Software Theory," *Journal of Systems and Software* **25**, p.219–222 (1994).

[Goguen79] J.A. Goguen and J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications," in *Proceedings Conference on Specifications of Reliable Software*, Boston (1979).

[Goguen93] J.A. Goguen, "Requirements Engineering as the Reconciliation of Technical and Social Issues," Technical Report, Centre for Requirements and Foundations, Programming Research Group, Oxford University Computing Lab, Oxford, U.K. (1993).

[Goguen94] J.A. Goguen, 1994. Private communication, via electronic mail.

[Habusha90] U. Habusha and D.M. Berry, "vi.iv, a Bi-Directional Version of the vi Full-Screen Editor," *Electronic Publishing* 3(2), p.3–29 (1990).

[Hoare71] C.A.R. Hoare, "Procedures and Parameters: An Axiomatic Approach," in *Symposium on Semantics of Algorithmic Languages*, ed. E. Engler, Springer-Verlag, Heidelberg, Germany (1971).

[Johnston71] J.B. Johnston, "The Contour Model of Block Structured Processes," in *Proceedings of ACM Conference on Data Structures in Programming Languages, SIGPLAN Notices* (1971).

[Lehman80] M.M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proceedings of the IEEE* **68**(9), p.1060–1076 (1980).

[Luqi90] Luqi, "A Graph Model for Software Evolution," *IEEE Transactions on Software Engineering* **SE-16**(8), p.917–927 (1990).

[Martin88] J. Martin and W.T. Tsai, "An Experimental Study in Upstream Software Development," Technical Report, University of Minnesota, Minneapolis, MN (1988).

[Parnas94] D.L. Parnas, "Software Aging," pp. 279–287 in *Proceedings of the Sixteenth International Conference on Software Engineering*, Sorrento, Italy (1994). Invited Plenary Talk.

[Srouji92] J. Srouji and D.M. Berry, "Arabic Formatting with ditroff/ffortid," *Electronic Publishing* 5(4), p.163–208 (1992).

[Sunshine82] C.A. Sunshine, D.D. Thompson, R.W. Erickson, S.L. Gerhart, and D. Schwabe, "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models," *IEEE Transactions on Software Engineering* **SE-8**(5), p.460–489 (1982).

[Tichy85] W. Tichy, "RCS—A System for Version Control," *Software—Practice and Experience* **15**(7), p.637–654 (1985).

[Wolfman90] T. Wolfman and D.M. Berry, "flo — A Language for Typesetting Flowcharts," pp. 93–108 in *Electronic Publishing '90*, ed. R. Furuta, Cambridge University Press, Cambridge, UK (1990).

SOFTWARE CHANGE-MERGING IN DYNAMIC EVOLUTION

David A. Dampier
Software Technology Branch, Army Research Laboratory
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, Georgia 30332-0800
dampier%airmics@gatech.edu

Valdis Berzins
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943
berzins@cs.nps.navy.mil

INTRODUCTION

Different variations of a software system are usually developed during software evolution. The need to apply a common change to each of these different versions is likely to occur during the lifetime of the system. It may also be desirable to combine the unique capabilities of two different versions into a new version. Because these software systems can be very large, tools that automatically perform these tasks are desirable. Change-merging provides the capability for such a tool[1].

EVOLUTIONARY PROTOTYPING

Rapid prototyping is an evolutionary approach to software development that was introduced to overcome the following weaknesses of traditional approaches:

1. Fully developed software systems that do not satisfy the customer's needs, or are obsolete upon release.

2. No capability for accurately evaluating real-time requirements before the software system has been built.

Rapid prototyping overcomes these weaknesses by increasing customer interaction during the requirements engineering phase of development, providing executable specifications that can be evaluated for conformance to real-time requirements, and producing a production software system in a fraction of the time required using traditional methods. Rapid prototyping allows the user to get a better understanding of requirements early in the conceptual design phase of development. It involves the use of software tools to rapidly create concrete executable models of selected aspects of a proposed system to allow the user to view the model and make comments early. The prototype is rapidly reworked and redemonstrated to the user over several iterations until the designer and the user have a precise view of what the system should do. In this approach to rapid prototyping, software systems can be delivered incrementally as parts of the system become fully operational [3].

EVOLUTION IN CAPS

The Computer-Aided Prototyping System (CAPS) is an evolutionary prototyping system designed

1. This research was supported in part by the Army Research Office under grant number ARO-145-91.

to prototype embedded, real-time systems [4]. CAPS consists of a set of prototyping tools connected together by a graphical user interface. One of these tools is an Evolution Control System that not only provides version and configuration control for the software system, but also provides project management control in the form of scheduling development tasks and automatic assignment of designers to those tasks. In the version and configuration control model for the system, development histories are represented using variations and versions. Each variation represents a parallel development history, and the versions comprise a series of increasingly accurate approximations to that particular variation. A variation/version number of 3.5 for a prototype means that this is the fifth version in the third variation.

CHANGE-MERGING

Change-merging is an integral part of the evolution methodology. During evolutionary development, multiple variations of a large system are likely to be developed. This can happen when independent development teams are working on different aspects of a system, or when alternate possible solutions to a problem are explored in different ways. Change-merging will enable these independently developed variations to be combined automatically, ensuring that the resultant system is semantically correct, with respect to all of the input variations, or it will report all conflicts preventing correct change-merging. This technology encourages the designer to explore multiple solutions to a problem, and to spread the development workload in a large project without the need for major efforts to subsequently integrate these independent efforts [3].

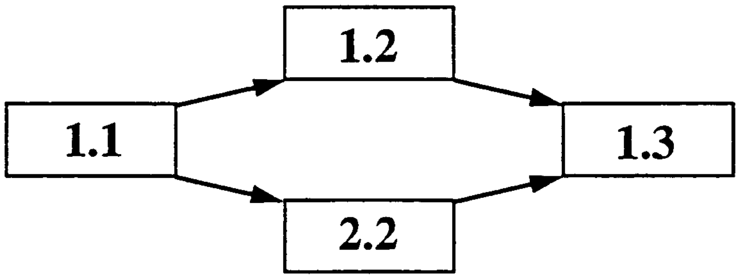


Figure 1: Change-merging two modified versions of a common base version.

Change-merging is a process by which significant changes between a base version of a software system and multiple modified versions can be isolated and combined into a single program as shown in Figure 1. As long as the changes do not conflict with one another, the result will be a program with the capabilities of all of the modified versions. Syntax-based change-merging methods like RCS and SCCS do this by manipulating text files [6,7]. They cannot provide any guarantee of correctness, however so semantics-based methods are needed.

CHANGE-MERGING IN EVOLUTION

Software change-merging can be used in several different ways in software evolution. As we already stated, it can be used to combine different changes to the same base program. It can also provide a way to update multiple existing versions of a program with a change made to the common base version as illustrated in Figure 2. In this example, version 1.1 is the base, versions 1.2 and 2.2 are the modified versions, and version 3.2 is the changed base. The result of each of these operations is a modified version updated with the common change. It can also be used to check consistency between independently developed versions. If a change-merge operation applied to two independently developed versions does not produce a conflict, then the versions are consistent.

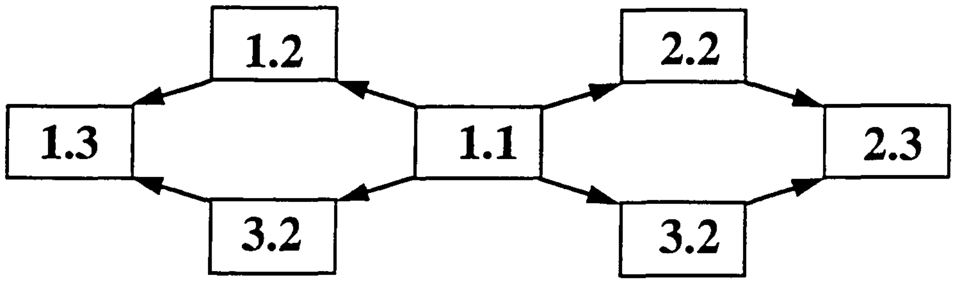


Figure 2: Updating multiple modifications with a change to the common base.

Another possible use of this technology is retracting changes from an evolution history. This idea is useful if after several iterations of the evolutionary process, the customer decides a feature of the software is no longer desired. Using change-merging it should be possible to automatically retract the change as long as the retraction does not cause a conflict in subsequent changes. The result of this operation would be a version that contains all of the capability in the most recent version of the system, except that contained in the retracted change, as shown in Figure 3.

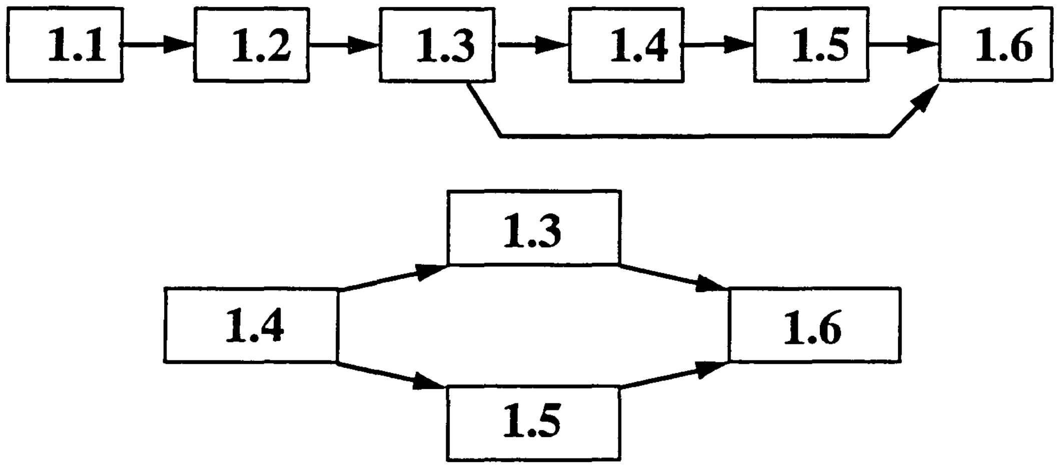


Figure 3: Retracting an earlier change from a subsequent version.

This example is designed to illustrate the removal of the change resulting in version 1.4 from version 1.5. Since 1.4 is the base version of the change-merge operation, the significant change from 1.4 to 1.3 is the retraction needed. This retraction must be preserved in the change-merged version, 1.6.

SUMMARY AND FUTURE WORK

We have developed a slicing method for change-merging prototypes written in PSDL, the prototyping language associated with CAPS [3]. This method will always produce a correct change-merged version if a conflict is not detected. Future work will include improving the resolution of the tool to prevent conflict reporting when no conflict exists, and trying to develop a change-merge method for other languages, perhaps Ada. Additional issues that are important for supporting evolution on a large scale include merging changes to system boundaries, module interfaces, and data representations. A key challenge is to formulate methods that can construct semantically well-formed merges in most of the cases where solutions exist that are not too far from those present in the versions provided as input, while remaining within the limits of practical computation times for large and realistic problems.

REFERENCES

1. Badr, S., *A Model and Algorithms for a Software Evolution Control System*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, December 1993.
2. Berzins, V., "On Merging Software Extensions", *Acta Informatica*, Springer-Verlag, 1986, pp. 607-619.
3. Dampier, D., *A Formal Method for Semantics-Based Change-Merging of Software Prototypes*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, June 1994.
4. Luqi and Ketabchi, M., "A Computer Aided Prototyping System", *IEEE Software*, March 1988, pp. 66-72.
5. Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real Time Software", *IEEE Transactions on Software Engineering*, October 1988, pp. 1409-1423.
6. Silverberg, I., *Source File Management with SCCS*, Prentice Hall, Englewood Cliffs, NJ, 1992.
7. Tichy, W., "Design, Implementation, and Evaluation of a Revision Control System", *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, September 1982, pp. 58-67.

AMPHION: Towards Kinder, Gentler Formal Methods

Richard Waldinger[2] and Michael Lowry[3]

Recom Technologies, M.S. 269-2
Artificial Intelligence Research Branch
NASA Ames Research Center
Moffett Field, CA 94035

Things are Awful

Although it is universally agreed that software is unreliable and takes too long to produce, formal methods have had little impact on the software development process. Why is that? Is it because most formal methods do not reduce time-to-market, but rather add another, time-consuming specification and verification phase to the end of the process? Is it because they require specifications to be written in a logical language that most practitioners find esoteric? Is it because practitioners must then guide a theorem prover to find a proof in a frightening logical system with an alien notation?

What's So Good About AMPHION?

AMPHION[1] is an application of formal methods that is free of this sort of drawback.

1. AMPHION is not an after-the-fact verification system—it is an automated software composition system that constructs provably correct programs from high-level components through deductive synthesis.
2. AMPHION does not require its users to construct a formal specification from scratch—it elicits an intuitively natural graphical specification gradually via a dialogue with a menu-driven user interface, thus combining advantages of visual programming and structured editors—but at a specification level.
3. AMPHION does not require its users to guide the proof process—it relies on an automatic theorem prover that has been tuned to yield accelerated performance for software composition.

1. AMPHION is the son of Zeus who charmed the stones lying around Thebes into place by playing his magic lyre to build the city's fortress wall.
2. waldinger@ai.sri.com; visiting from the Artificial Intelligence Center, SRI International.
3. lowry@ptolemy.arc.nasa.gov

Software of Astronomical Complexity

AMPHION is a domain-independent and language-independent architecture; it is specialized to a particular application through an application domain theory. Its first application has been to develop software to perform computations of the observation geometries for interplanetary missions for mission planning and data analysis. That software is composed from subroutines in SPICELIB, a JPL subroutine library written in FORTRAN-77. AMPHION is now also being applied to other NASA domains, including space shuttle flight planning and the composition of software for numerical aerodynamic simulation.

AMPHION is more than a research prototype: it has already undergone substantial testing with planetary scientists over a period of six months and is currently undergoing further enhancements in preparation for distribution to the large user community for SPICELIB. The specification acquisition component is easy to learn: users are able to develop their own specifications after only an hour's tutorial. Observations over six months indicate at least an order of magnitude improvement for specification development over manual program development. Programs which would take the better part of a day to develop for someone only casually familiar with the subroutine library can be specified in fifteen minutes after the tutorial introduction to AMPHION. Experienced AMPHION users can develop specifications in five minutes for programs that would take the subroutine library developers an hour to code manually. AMPHION's program synthesis component is robust and efficient, and appears to be the first use in practice of totally automatic deductive program synthesis. AMPHION synthesizes, from specifications, one- to two-page programs consisting of one- to three-dozen calls to SPICELIB subroutines in just a few minutes. In over a hundred programs generated by AMPHION to date for the NAIF domain, the CPU time to synthesize a program never exceeded five minutes.

To illustrate, let's consider a small sample problem: A

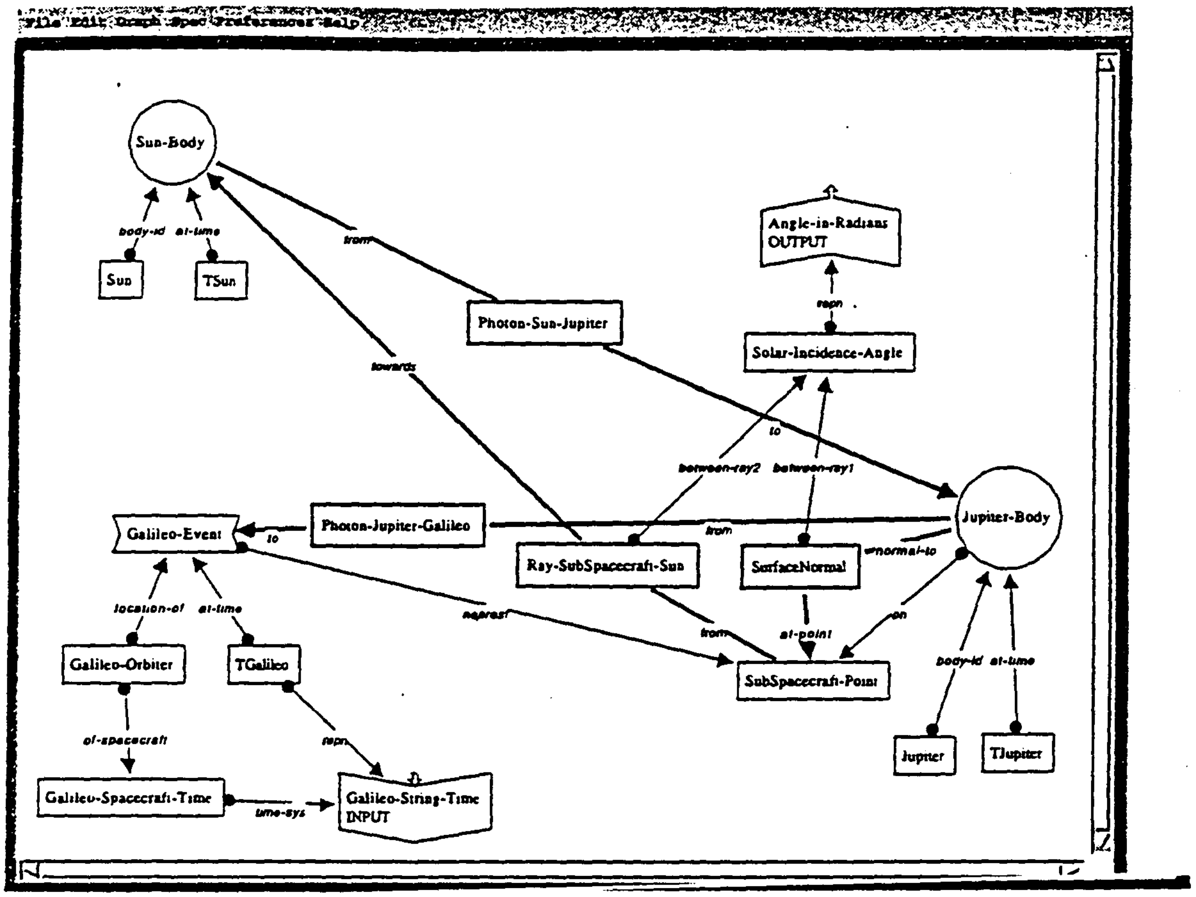


Figure 1: Diagram for solar incidence angle developed interactively with AMPHION.

```
      SUBROUTINE SOLAR ( GALILE, ANGLEI )
C     Input Parameters
      CHARACTER*(*) GALILE
C     Output Parameters
      DOUBLE PRECISION ANGLEI
C     Function Declarations
      DOUBLE PRECISION VSEP
C     Parameter Declarations
      INTEGER JUPITE
      PARAMETER (JUPITE = 599)
      INTEGER GALIL1
      PARAMETER (GALIL1 = -77)
      INTEGER SUN
      PARAMETER (SUN = 10)
C     Variable Declarations
      DOUBLE PRECISION RADJUP ( 3 )
      DOUBLE PRECISION E
      DOUBLE PRECISION PVGALI ( 6 )
      DOUBLE PRECISION LTJUGA
      DOUBLE PRECISION V1 ( 3 )
      DOUBLE PRECISION X
      DOUBLE PRECISION PVJUPI ( 6 )
      DOUBLE PRECISION LTSUJU
      DOUBLE PRECISION MJUPIT ( 3, 3 )
      DOUBLE PRECISION V2 ( 3 )
      DOUBLE PRECISION X1
      DOUBLE PRECISION DV2V1 ( 3 )
      DOUBLE PRECISION PVSUN ( 6 )
      DOUBLE PRECISION XDV2V1 ( 3 )
      DOUBLE PRECISION V ( 3 )
      DOUBLE PRECISION N ( 3 )
      DOUBLE PRECISION PN ( 3 )
      DOUBLE PRECISION DV2N ( 3 )
      DOUBLE PRECISION XDV2N ( 3 )
      DOUBLE PRECISION DXDV2V ( 3 )
      DOUBLE PRECISION XDXDV2 ( 3 )
C     Dummy Variable Declarations
      INTEGER DMY10
      DOUBLE PRECISION DMY20 ( 6 )
      DOUBLE PRECISION DMY60 ( 6 )
      DOUBLE PRECISION DMY130
      CALL BODVAR ( JUPITE, 'RADII', DMY10, RADJUP )
      CALL SCS2E ( GALIL1, GALILE, E )
      CALL SPKSSB ( GALIL1, E, 'J2000', PVGALI )
      CALL SPKEZ ( JUPITE, E, 'J2000', 'NONE', GALIL1,
                   DMY20, LTJUGA )
      CALL VEQU ( PVGALI ( 1 ), V1 )
      X = E - LTJUGA
      CALL SPKSSB ( JUPITE, X, 'J2000', PVJUPI )
      CALL SPKEZ ( SUN, X, 'J2000', 'NONE', JUPITE,
                   DMY60, LTSUJU )
      CALL BODMAT ( JUPITE, X, MJUPIT )
      CALL VEQU ( PVJUPI ( 1 ), V2 )
      X1 = X - LTSUJU
      CALL VSUB ( V1, V2, DV2V1 )
      CALL SPKSSB ( SUN, X1, 'J2000', PVSUN )
      CALL MXV ( MJUPIT, DV2V1, XDV2V1 )
      CALL VEQU ( PVSUN ( 1 ), V )
      CALL NEARPT ( XDV2V1, RADJUP ( 1 ),
                    RADJUP ( 2 ),RADJUP ( 3 ),N, DMY130)
      CALL SURFNM ( RADJUP ( 1 ), RADJUP ( 2 ),
                    RADJUP ( 3 ), N, PN )
      CALL VSUB ( N, V2, DV2N )
      CALL MTXV ( MJUPIT, DV2N, XDV2N )
      CALL VSUB ( V, XDV2N, DXDV2V )
      CALL MXV ( MJUPIT, DXDV2V, XDXDV2 )
       ANGLEI = VSEP ( XDXDV2, PN )
      RETURN
      END
```

Figure 2: SOLAR program generated by AMPHION from Figure 1.

planetary scientist working on the Galileo mission to Jupiter wants to compute the solar incidence angle at the point on Jupiter's surface closest to Galileo—the subspacecraft point.

Through a dialogue with the user interface, the scientist comes up with the graphical specification illustrated in Figure 1. The figure is geometrically suggestive—the circle in the upper-left corner corresponds to the space-time location of the sun, the circle at the right to that of Jupiter, and so forth. Users can even customize the appearance of items in a diagram; e.g., use bitmap pictures for planets. AMPHION's specification vocabulary for this domain is at the level of abstract Euclidean geometry (e.g., points, rays, ellipsoids, and intersections) augmented with astronomical terms (e.g., photons and planets). Note there is no mention of implementation-level coordinate frames, units, and so on; except in defining representations for inputs and outputs. Implementation-level constructs are introduced during program synthesis. This graphical specification is automatically translated to a formal problem description in first-order logic (augmented with the lambda calculus). After proving a theorem, AMPHION composes the FORTRAN-77 program in Figure 2. This program was generated in 96 seconds on a Sparc 2.

What's the Trick?

Automatic theorem provers have been applied to software development problems for quite some time without having much practical impact. Why has AMPHION been able to construct software of a practical level of complexity?

First of all, AMPHION does not attempt to build software from the primitive instructions of a programming language. It is content to start from high-level components in a mature software library. Each component already embodies a good deal of domain expertise and programming knowledge—AMPHION is merely gluing it together.

Secondly, AMPHION's theorem prover does not attempt to maintain equal competence in all subject areas. AMPHION invokes the theorem-prover SNARK, which has been developed at SRI by Mark Stickel particularly for applications in software engineering and artificial intelligence. SNARK allows us to provide domain-specific control strategies, which guide its attempt to discover a proof. For example, for the planetary astronomy domain SNARK employs a special strategy, devised by Thomas Pressburger, that would only be appropriate for software composition. This gives it a sense of direction atypical of domain-independent theorem provers. In particular, the strategy guides SNARK from abstract, specification-level constructs towards concrete, implementation-level constructs.

How's it Work? Here's the Skinny!

The user of AMPHION engages in a dialogue with the menu-driven interface, developed by Ian Underwood and Andrew Philpot, that elicits the graphical specification; the user does not need any prior knowledge of the specification language or the software component library. Specifications developed with the interface are correct with respect to syntax and type.

Once the user is satisfied with the specification, it is subjected to a preliminary semantic analysis that often reveals common specification errors. It is then rephrased as a mathematical theorem, which states the existence of an output entity that satisfies the specified conditions.

The theorem is then sent to the theorem prover. SNARK employs a classical first-order logic but is restricted to be sufficiently constructive that, in proving the existence of the desired output entity, it is forced to indicate a method of finding it. That method becomes the basis for a program to compute the output, which is extracted automatically from the proof.

The proof is conducted in an application domain theory that provides the knowledge on which the software depends. The specifications of the components in the library, the constructs of the specification language, and the properties of the application domain necessary for gluing the components together are all represented by axioms in the domain theory. This theory also determines the options offered to the user by the graphical interface, which is itself domain-independent.

The program extracted from the SNARK proof is in an applicative language; it is translated into the desired target language, such as FORTRAN, ADA, or C; in a subsequent transformation phase, which is relatively quick and straightforward.

The proof from which the program is extracted establishes the correctness of the program with respect to the user's specification, provided the domain theory is accurate. For high-assurance applications, one would require that the correctness of the components must also be verified, perhaps by the same process.

What More Could You Want?

If a piece of AMPHION-constructed software needs to be enhanced or modified, the changes are made at the specification level. In fact, users have most often found it convenient to develop graphical specifications for new problems by revising existing specifications for similar problems. The abstract graphical notation makes it much easier to identify the required modifications than tracing through dependencies in someone's code. AMPHION's graphical editing operations facilitate making the changes. AMPHION retains a library of specifications that can be used for the purpose of specification reuse and modification. No attempt is made to retain the proof or the previous program—the proof process is fast enough so that it may be reconstructed entirely from scratch. In this way there is no danger that in modifying the program we may be compromising its correctness.

Some aspects of the domain theory development and extension were relatively straightforward. For example, during a visit in September of 1993, the head of the JPL group that developed SPICELIB wanted to specify a problem involving a new feature that was not yet part of the domain theory: the pole of a planet. In fifteen minutes we were able to add in the declarations and axioms to the domain theory that were needed, and then demonstrated AMPHION generating a FORTRAN-77 program for a specification involving the pole of a planet. We anticipate developing tools that will empower domain experts to make such incremental extensions to a domain theory themselves.

In the planetary astronomy application, software was constructed from a single subroutine library. However, different routines would assume different coordinate systems or different units of time measurement. The AMPHION user could compose these routines and remain completely oblivious to differences in representation. This suggests that AMPHION may be equally capable of composing software from diverse machines or environments, even if the user is ignorant of discrepancies in representation, languages, and ontology of these environments.

Further Reading

A Formal Approach to Domain-Oriented Software Design Environments,
M. Lowry, A. Philpot, T. Pressburger, and I. Underwood, KBSE'94, September 1994.

AMPHION: Automatic Programming for Scientific Subroutine Libraries,
M. Lowry, A. Philpot, T. Pressburger, and I. Underwood, ISMIS'94, October 1994.

Deductive Composition of Astronomical Software from Subroutine Libraries,
M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood, CADE-12, June 1994.

Fundamentals of Deductive Program Synthesis,
Z. Manna and R. Waldinger, IEEE Transactions on Software Engineering (18) 8, August 1992, 674-704.

Design of Languages for Multimedia Applications Development

Jacob T. Schwartz and W. Kirk Snyder
New York University Center for Digital Multimedia
August 1994

1. Introduction.

The design of programming languages intended to support the development of multimedia applications raises interesting new challenges for the language designer. The multimedia world is very different from that contemplated in traditional language design. Images are central data objects; video and sound must also be accommodated. Interactivity is a second central issue. As in all language design efforts, a key strategic goal is to develop conceptual models of the essential semantics of a language's intended application area which are powerful enough to allow the designer to 'put it all together'. That is, one aims to design a framework which embodies just the right tradeoffs between featureless generality and limiting specificity, in a way which brings powerful, intuitive capabilities together in an environment that facilitates their easy use. In abstract terms, the multimedia area differs from the would of conventional programming in the greatly expanded role of large data objects such as images and sound which need to be 'sculpted' within an interactive environment rather than described numerically. The working environment can no longer be the simple sort of text editor that suffices for conventional programming; a multimedia 'Language' must be as much an advanced interactive environment as a conventionally structured programming language. Thus both aspects of multimedia authoring tools must be designed together. In the multimedia area one must also aim to accommodate large projects and large data sets, and to support cooperative work by substantial groups of developers. Multimedia therefore challenges the language designer in new ways and is suggestive of ideas unlikely to arise in connection with more conventional applications areas.

The academic world has not yet contributed substantially to this industrially driven area. However, a wide variety of commercial tools, embodying an interesting variety of design approaches and suggestive of paradigms that need to be refined and generalized, have appeared. Some of the most significant of these are:

(i) Macromind Director. This widely used Macintosh/Windows authoring/animation tool is organized around a 'musical score' notion allowing graphics and interaction widgets to be constructed using a small, well-integrated set of sculpting tools. The current version 4.0 of this system also incorporates a general purpose programming language ('Lingo') which supports creation of interactivity via a system of event-handlers. Lingo also supports various standard procedural constructs, up to and including elementary list facilities.

(ii) Visual Basic. This is much more a standard programming language than Director is, but nevertheless is organized around much the same event-handler notion as Director. Visual Basic provides various significant capabilities not available in Director, e.g. access to full database systems and easy linkages to C and to Windows inter-application linking capabilities such as OLE, which are apt to be of particular importance in the multimedia world.

(iii) Many interesting 'elemental' or 'asset preparation' tools, which serve for the preparation and organization of images, audio, video, MIDI music, and related forms of data, have become available to the multimedia applications designer. These include Photoshop (Image preparation), Premiere (video editing), Illustrator (graphics), Strata Studio Pro (3-D modeling and animation), Sound Edit (Macintosh sound editing), Elastic Reality (Image morphing), Virtus Walkthrough (3-D modeling tool supporting real-time virtual reality walkthroughs), Pixar Typestry (Specialized font-based 3-D animation tool), Streamline (conversion of bitmapped art into line drawings), ScreenCam (screen-clip utility), Vision (Midi music development and acquisition), and Debabelizer (color palette reconciliation tool). The foregoing list emphasizes commercial Macintosh applications; however, many of the same tools are also available in the Windows environment. Generally comparable tools are available on other higher-end platforms, e.g. Sparcs and Silicon Graphics machines, and on platforms ranging from the mid-range systems on up to functionally comparable but higher performance systems which only run on specialized systems priced in the $100,000 - $1,000,000 range.

All these applications are sources of new ideas for multimedia work-environment design. A particularly interesting aspect of many of them are the sets of 'widgets' through which much of their functionality is made available. 'Widgets' are simple or complex graphic entities which respond to user mouse clicks and other gestures by changing their graphical appearance in various helpful ways, while at the same time performing associated but invisible data manipulations. To take just one interesting example, a significant widget used in 'Director' is its 'score', which incorporates some of the 2-D layout of a musical score (but as time versus graphical element or 'sprite' instead of the time versus pitch and instrument). This widget allows very natural gestural representation of many important multimedia collaging and animation operations.

2. The 'Panel' Language.

At the New York University Multimedia Center we have recently begun an effort to design and implement a new multimedia development language tentatively named 'Panel'. This is much influenced by 'Director', but aims at a major strengthening of its capabilities, especially its ability to support large educational projects that must incorporate complex simulations and interactivity. The remainder of this paper will review various aspects of the design of this system, with something of an emphasis on motivations for significant design decisions. One such design criterion for Panel is that it be self-describing, in the sense of providing facilities powerful enough for easy definition, via a combination of standard programming and gestural description ('sculpting'), of all the widgets that the language will use.

(i) The 'Internal' part of the Panel language will be close to an extended version of the SETL very high level language. In the presence of the extensions described below, this should support complex simulations and interactivity well. However, the language will be extended to support various multimedia objects, e.g. sound files, pictures consisting of drawn and bitmapped graphics, and video directly.

(ii) It is assumed that Panel will exist in an operating environment in which other like programs may be running. All of these will be able to intercommunicate via a system-maintained set of message queues, which will also serve as a source of standard 'events'.

(iii) It also assumed that Panel will exist in an operating environment which connects it to various local area nets and to Internet. This is reflected in the availability of various high-level services, e.g. an FTP service having the form of a function call identifier_atom := FTP_get(internet_file_name,local_file_name,password)

which either copies the file located by the internet_file_name into the file located by the local_file_name and returns a message of the form

[transfer_successful,identifier_atom],

or fails.

(iv) An object construction. In many cases it is useful during graphic (and other forms of real-time) experimentation to 'attach' some of the constants of a procedure (or more generally an object) to graphical widgets (e.g. sliders), allowing them to be varied easily for experimental purposes. Panel's internal language will provide a general way of doing this, without restricting the widgets used to any specific form. This idea will be supported by one of a small group of related extensions to SETL, which we will now outline. These extensions will be realized as extensions to the SETL var declaration, namely

static var v_1,v_2,...; pointer var v_1,v_2,...; instrumented var v_1,v_2,...;
persistent var v_1,v_2,...;

The meaning of these declarations is explained just below. In some cases variables will be declarable with several of these qualifying suffices, e.g.

static instrumented var v_1,v_2,...;

and so forth.

The meaning of these declarations is as follows.

static var v_1,v_2,...;

Has much the force of the ALGOL own declaration.

pointer var v_1,v_2,...;

is relevant for the declaration of instance variables within objects, and states that changes in the value of any of the variables listed does not change the logical identity of the affected object instance, i.e. does not imply any copy. This in effect suspends the normal SETL insistence on pure value semantics, and gives instances of the object containing the declaration 'pointer semantics' in regard to the variables declared as pointer var.

The considerably more novel declaration

instrumented var v_1,v_2,...;

changes the semantics of left- and right-hand occurrences of the variables v_1,v_2,..., provided that they have also been subject to 'procedure assigning calls' using the special built-in function set_procedure(v_j,lhs,rhs);

In this call it is expected that lhs is either a procedure of one argument or some other value, and that rhs is either a procedure of zero arguments or some other value. Moreover, in the presence of the declaration and of the call shown, right hand occurrences of v_j are treated as occurrences of the construct that would otherwise have to be written as

```
if is_procedure(rhs) then rhs() else v_j end if.
```

Likewise left hand occurrences of v_j := x are treated as occurrences of the construct that would otherwise have to be written as

```
if is_procedure(lhs)
   then lhs(x);
   if rhs = OM then v_j := x; end if;
else
   v_j := x;
end if;
```

Thus to assign a right-hand procedure to a variable v that has been declared instrumented one simply sets rhs equal to the procedure; and then to 'freeze' the value one executes

```
procedure freeze(v,lhs);  -- the lhs will be retained
  set_procedure(v,OM,OM);
  v := rhs();
  set_procedure(v,lhs,OM);
end freeze;
```

similarly for the left-hand side.

The declaration

```
persistent var v_1,v_2,...;
```

makes a simple form of persistent storage available in conjunction with the present SETL 'library and package' system. This declaration should occur within a SETL package or package body declaration, e.g. in the form

```
package my_pak;
    persistent var v_1,v_2,...;
end my_pak;
```

The associated package body may or may not contain additional code.

In the presence of this declaration any changes in the values of the variables v_1,v_2,... made by a program that uses the package, e.g. by a program

```
program my_prog;
   use my_pak;
```

```
                    end my_prog;
```

is written (at program exit, but not before, and in particular not if the program aborts before exit) to the library file housing the package, making these values available for subsequent use by any programs that subsequently use the package. This creates a convenient facility for storage of large multimedia objects by edit programs that can create them, followed by retrieval, as 'resources', by applications that need them.

Widgets.

Panel will represent widgets procedurally, specifically by procedures which can also access and display graphic elements using the standard primitives and conventions described below. (Note however that in favorable cases it will be possible to compose these procedures very easily from lower level routines using auxiliary widgets.) Any widget must therefore have some way of:
(i) acquiring, and then accessing, all the data needed to define any graphic object or objects which it will display;
(ii) receiving notification of the events to which it must respond.

The widget will of course be an object instance OI, and for (ii) we simply propose that all widgets must support a method OI.react(event), where 'event' is a standard form event message representing the events acknowledged by the system, e.g. mouseDown, mouseUp, mouseRoll, clockTick, keyDown, etc. It is then the responsibility of the widget to respond appropriately.

In regard to (i) we note that, when ready to run, the widget can store all the PICT and other data it needs in a set of persistent variables of the widget class. To set these variables initially, the widget should provide a method

```
initialize(file_name);
```

which receives a binary file name as parameter, reads the corresponding file to find all the subsequently persistent data it needs, and installs this data in the appropriate persistent variables. Note that this 'initialize' procedure only needs to be called when the widget design has been changed; at other times the widget will find all the graphic and position data it need in a set of persistent variables attached to an appropriate one of its defining packages.

Conventions for Event handling in widgets; Hierarchically Designed Widgets.

It will often be effective to construct widgets hierarchically, out of subwidgets which are themselves constructed out of sub-subwidgets, till at the bottom level only simple graphic elements need to be constructed. The following issues need to be faced in this connection: (i) How to get the graphic appearance required, and in particular how to arrange (and possibly resize) the separate widgets in appropriate geometric relationship to one another; (ii) How to transmit the events to which the widget must respond to the subwidgets which will actually generate the response. As to generating widget-associated graphics we propose that a procedural approach be regarded as basic. I.e. a widget, on receiving the parameters p_1,..,p_n needed to define its graphical appearance, will use them to make whatever top level calculations are necessary, following (or during) which it call recursively on every one of its subwidgets, each of which will return its own representing graphic. (As explained in more detail in the following section, each such 'graphic' is actually a tree of graphic images with their 'inks', each such image being logically resident in its own bitplane, until the ultimate combination of all of them into a final image to be displayed to the user.) These graphics will then be moved to their appropriate relative positions and a vector of graphics returned.

Each widget is also obliged to support a OI.react(event) call. For hierarchically structured widgets, the skeletonized structure of this 'react' procedure will be approximately as follows: The top level widget code will decide procedurally which of its subwidgets is actually to handle the call. It will then initialize for the call, next execute the call, and finally respond in appropriate manner to any values returned by the call.

The weakness, yet at the same time a strength, of this approach is that it makes each top level widget responsible for deciding, in some unspecified procedural manner, which of its subwidgets should respond to each incoming event and how each such subwidget should see the event. It is however easy to suggest conventions which 'decentralize' this decision making in a standard, intuitive, and convenient way. Specifically, we can agree (for 'ordinary' widgets, not necessarily for all) that mouse events (which normally are mouseUps, mouseDowns, and MouseEnters) are always routed to the frontmost non-disabled (hidden) bottom-level graphic, in the back-to-front order in which we will always arrange graphics (see below for additional detail), at the point of the mouse event. (A MouseEnter event takes place when the cursor crosses a region boundary; the nominal place of the event is the place of this crossing). It is then the responsibility of the frontmost graphic object receiving the event to respond to the event, and no other object will respond unless this object either has no handler for the event or if its handler decides to pass the event along (a primitive operation is provided for this purpose.)

On the face of it this convention only allows handlers to be associated with bottom-level graphics, not with composite graphics composed hierarchically out of lower level elements. However, if one wants to associate some action with one or more higher level entities, it is only necessary to cover the full geometric extent of each of these with a transparent graphic. This can respond, and then pass the event through to the widget's subcomponents for additional processing if more needs to be done.

Since the graphic environment we propose (described below) is one in which each graphical object is nominally written to its own graphics plane (before these are all combined into one image by a the system's 'rendering engine') it is easy to set up this association. The routine that draws graphics can be supplied with a triple of handler routines for each graphics plane, each of which should also be marked with the extent of the graphic it contains (as a 'region' represented in some suitable condensed format, so that the sensitive portion of a graphic can be of arbitrary shape.) The draw routines in total then need to write this information, in some standardized form (say as a map from image plane number to sensitive region and handler tuple) to a variable accessible to the event interpreter for the widget. This interpreter can then search this data structure to determine the handler to which the event should be dispatched.

Here as later we meet a structuring convention which seems highly acceptable in the sense that most ordinary screen interaction management conforms to it, and also in the sense that it can be evaded when necessary just by placing a transparent cover over the whole screen which can respond in any fully procedural, non-standard way desired.

More Details Concerning the Panel Graphic Environment.

This section give more details of the Panel graphic environment by spelling out some of the conventions that the crucial objects used in this environment must satisfy. A central aim of our design is to make it easy to create both highly procedural graphical objects (which are useful for creating all kinds of screen widgets) and collages of simple, non-interactive picture-like objects. That is, we want it to be easy both to attach images (via an appropriate degree of sculpting) to procedures written as code of ordinary form, and to sculpt relatively static, Director-like collages of images and text. A related aim is to ensure that all the sculpting widgets used in the Panel implementation can be constructed using the set of draggable graphic objects provided by Panel itself. A third aim is to ensure that the collection of graphical objects provided can be extended indefinitely by adding high efficiency, C-written implementations of new graphic primitives as these are invented.

Toward these ends, the following conventions are proposed:

(i) An inked color bitmap placed at a specific (x,y) position on a graphic plane is called a placed picture. Since Panel's render routine will deal just as well with nested tuples of such objects as with simple placed pictures, we will sometimes use the term 'placed picture' to refer to what, strictly speaking, are nested tuples of placed pictures.

(ii) Placed pictures will always be generated from image objects. Each such object O must support a standard parameterless method O.get_image(), which returns a nested tree of tuples whose bottom level elements are placed pictures. The data internal to these objects can be manipulated by additional calls, whose details can vary from object to object. Changing the internal data of O will normally change the value of O.get_image().

In addition, each image object O must support a method O.get_frame(), which produces some appropriate form of simplified graphical representation (a 'drag frame') of the true image produced by O.get_image(). The image returned by O.get_frame() should be simple enough to allow dragging in real time, but should represent O.get_image() faithfully enough to suggest its eventual shape and position clearly. (A familiar example is the use of 'dotted boxes' to represent rectangular bitmaps and windows while these are being dragged.) If O.get_image() can be calculated in real time, it is best for O.get_frame() to be identical to O.get_image().

(iii) Image objects are required to be translatable, in that they must have a standard set_position method:
OI.set_position(x,y)
The screen image of OI which 'render' produces immediately after a call OI.set_position(x,y) should relate in the appropriate geometric way to the screen image of OI immediately after a call OI.set_position(x',y',) with different parameters.

(iv) The required O.get_image() call should actually return a bit more information than just a placed picture; namely information that can be used to optimize the 'render' routine which composites any tree (or sequence of trees) whose bottom level elements are placed pictures into an overall screen image. Ignoring these optimization considerations for the moment, we can describe the workings of 'Render' as follows:

(a) Walk the tree (or sequence of trees) in left-to-right order, converting its collection of leaves into a linear sequence of leaves, each with an [x,y] position, and with an ink.

(b) Starting from a background of known color, apply the inks separately in each image plane, to calculate an overall image I.

(c) assemble all of these images I into a single screen image.

Like all 'structuring' conventions in programming languages, the graphical conventions just outlined restrict the generality of the system being designed in order to make it more intuitive and simplify its use. Conventions of this sort succeed if the freedom they give up is undesired in most cases because the structures they create generally lie close to the paths that even an unconstrained design would actually take. The specific conventions outlined imply that the Panel graphic environment must always be viewed as an ordered sequence of 2-D graphical objects seen in some back to front order and subject to various quite general rules of translucency. This is a convention used comfortably in may 2-D graphics systems including Director, an moreover the limitation of flexibility that it seems to imply is easily overcome since it allows an entirely general graphic, e.g. a 3-D animation projected down to two dimensions for screen display, to be written any of the graphic planes superimposed to make the final image seen by the user.

We construct the sequence of graphics to be superimposed by 'Render' ass a tree rather than a simple sequence in order to give the graphic environment a fully recursive character. This allows nested graphic structures of arbitrary complexity, e.g. complex subwidgets of even more complex widgets, to be returned by recursive construction routines which can regard them as graphic totalities.

Efficiency considerations.

Rendering of any particular image object may either be efficient enough to be pleasantly immediate in real time, or less efficient. As inefficiencies mount one goes from situations in which the rendering takes a few seconds to situations in which rendering operations run for hours or days. If some comprehensible representation an image object can be rendered in a few tenths of a second as its parameters vary, the object is usefully 'draggable' by the mouse, as above. Even if this is not the case, when simple sequences of images are wanted they can be generated off-line and then turned into static arrays of pictures. As in the rendering of morphs or of a 3-D fly-through into QuickTime movies, this may expand a small amount of data into a very large data mass, which is one of the reasons why static rendering is undesirable if it can be avoided; another reason is that multidimensional (rather than 1-dimensional) interactivity is lost.

Some Examples.

To gain confidence in the broad utility of the 'frontmost object responds' convention proposed above, we will now outline the way in which various well-known widgets and subwidgets can be constructed using this convention.

Buttons are very simple: they are just graphics with associated response routine. Buttons (like check-boxes) which change their appearance when clicked simply need to substitute one graphic for another and redraw the screen. Groups of radio buttons are merely buttons any one of which may which change appearance when some other button in the group is clicked.

Simple menus can be represented either as a graphic sensitive to mouseRoll events (so that the usual 'highlight bar' can move along with the mouse to indicate the current selection), or as a graphic covered by a transparent graphic consisting of multiple rectangles, one such rectangle covering each menu line, with omitted, 1-pixel wide strips separating them, so as to generate mouseEnter events whenever the cursor moves between menu lines, allowing the highlight to be shifted as necessary. To get the standard Macintosh menu action, in which the menu remains visible even when the cursor leaves it (but disappears when the cursor enters the menu bar), a transparent graphic covering the whole screen should be placed immediately behind the menu (as usual, we assume that any widget has been moved (or copied) to frontmost position whenever it is active.) The presence of this transparent 'cover' prevents other items present on the screen from reacting as long as the menu remains open.

User draggable graphics (2-D sliders) are graphics sensitive to mouseRoll events, which record the position of the mouse when first clicked and then continually reposition themselves to maintain a fixed position relative to the mouse. Like menus, draggable graphics should place a transparent cover across the whole screen, since they may need to recapture the cursor if it has move out of them between successive mouse roll events. When employed as a drag handle attached to some more complex object, they may not only move their own positions but also change the appearance and position of other graphics. Draggable graphics can also record the history of their own motion in some auxiliary data structure available to other procedures.

1-D sliders, rotating knobs, graphic trackballs, etc. are much like 2-D sliders, except that they position themselves in some graphically constrained way involving a simple or not-so-simple calculation based on mouse position.

Graphic objects which need to respond to all clicks sufficiently near their extent can simply cover themselves with a slightly larger transparent graphic defining this sensitive region. Groups of objects which need to respond differently depending on which object is closet to the point of click can cover the whole area to which the group is sensitive with a single transparent graphic, which can then respond by passing the event on to the closest object in the group.

A standard paint window consists of a primitive color bitmap with an invisible draggable graphic (the 'brush', 'eraser', 'paint bucket', etc.) of selectable shape (and possibly other attributes) which modifies the bitmap in some specified way when dragged over it or clicked on it.

Text with hotwords consists of text graphically expanded by use of appropriate fonts, stylings, and color conventions in which the collection of words marked 'hot' is covered by a collection of rectangles (probably grouped into a single 'broken rectangle') whose position and geometry is derived automatically from the string contents and styling of the text. Each 'hotmark' should then carry an indication of the code to be invoked when the hotword is clicked. The superimposed broken rectangle will respond via a case statement in which all these code fragments are grouped together.

The availability within Panel of powerful means for constructing widgets of the sophistication outlined above will tend to give Panel programming a somewhat different flavor than that typical for Director programming. Often, where Director uses multiple successive or even separated screens, Panel will instead display a widget of changing appearance against a static background. For example, a 'jumping' text widget, which displays successive marked sections of a continuous text too large to be appropriate for single-screen display may be seen against a fixed background, or 'reveal-and-conceal' text which covers marked parts of itself with translucent rectangles may be used. This will give Panel programming something of the flavor familiar from Hypercard, though of course all the resources of Director will also be available.

Construction by Pure Sculpting.

Whenever the layout and appearance of an hierarchically structured widget is not too elaborately variable or repetitive construction by pure sculpting may be a feasible alternative. To convince ourselves of this, we have only to propose sculpting conventions capable of defining and collaging the basic graphic elements and relationships mentioned in the preceding paragraphs. Sculpting operations can use a set of simple auxiliary tools, somewhat resembling the columns of the Director 'score', to support a useful kind of 'layered' subwidget construction. We will call these tools 'layer columns', and imagine them to have the appearance shown in Figure 1 below. Such layer columns are used to arrange the subwidgets of a hierarchically constructed widget in back-to-front order. As shown in the figure, the parts of a layer column are its header (which assigns it a string name), followed by successive rows designating its parts. Each part has an optional part name (if this is left blank, a number is used instead), a visibility indicator (used during editing, to hide parts so as to avoid clutter), a subhierarchy-open indicator) to hide and show the naming details of subobjects, and a subwidget name, which must either indicate that the part is a primitive widget of some kind or must be identical to the Column name of some other column in the hierarchy being constructed. An auxiliary 'cast' widget showing helpful thumbnails is assumed to be available; as primitive widget components are constructed, by painting or in some other way, they are assigned positions in this cast. To begin constructing a hierarchical widget using 'layer columns' one opens one or more such layer columns, gives each of them a unique identifying name, and then installs the needed primitive widgets into them. As in Director, this is done by clicking on an unoccupied row in a layer column to identify it, and then dragging the desired primitive from the cast to some desired position on stage. Bottom level layer columns will consist entirely of primitives; higher-level layer columns will reference lower level columns, introduced into them by clicking on an unoccupied row in the higher-level column and dragging the desired lower-level object on stage, using a drag starting in its name field. Interaction with the parts of widgets makes use of the 'frontmost active object' responds convention described above, and of a Director-like convention for assigning code blocks to the mouse events to which graphic objects will ordinarily be sensitive.

An Algebra for Ontology Composition

Gio Wiederhold

ARPA and Stanford University

July 1994

INTRODUCTION

To compose large scale software there has to be agreement about the terms, since our models depend on symbolic linkages among the components. In modestly-sized systems, we implicitely count on such an agreement. Within a specific domain terms are indeed likely to be consistent, so that specifications can be developed from English (or other natural) language source documents. When combined with a coherent framework we have the underpinnings for a Domain-Specific-Software Architecture (DSSA). In this abstract we propose extending concepts used in object-based structural algebras and DSSA research to a knowledge-based algebra, suitable for composing larger systems that span multiple domains.

The principal operations in the algebras are simple and provide for selection from the objects in the source domain space and placing them into new domains that represent the information needed for the composed results.

1. BACKGROUND

Divide-and-conquer is an essential approach in science and technology, and in large software systems as well. Early manifestations of division of tasks in software were scientific subroutine libraries, since their development and evaluation required uncommon rigor. These libraries grew to encompass statistical procedures SAS [Rose:83], and specialized libraries serve diverse domains as planetary navigation NASA [Acton:93] and Graphical User Interfaces. Today commercial firms or funded service agencies provide most of such libraries.

An alternative approach is the development of packages, which are not intended to be integrated into larger suites. These were also popular in the statistical domain, as BMD [Dixon:69], but have been largely supplanted by composable routines, which allow the application of statistics to a variety of application domains.

In this abstract we consider the construction of software from autonomous modules, *megaprogramming* [Wiederhold:92]. We refer to the scope of autonomous modules as their domain, and to the terms used to describe items and their relationships in a domain as their domain `ontologies`. While the terms in these ontologies are often only manipulated in paper form or perhaps as IDEF [Loomis:87] documents, we believe that the contents of ontologies warrants formal manipulation if reliable systems are to be composed.

2. DOMAIN SPECIFIC KNOWLEDGE

Object technology has blosomed due to the incorporation of semantics within the units that the software and retrieval strategies deal with. The definitions that make retrieved objects coherent are particular to a specific application area and its domain. The focus of research in Domain-Specific-Software-Architecture (DSSA) has been the acquisition of knowledge in a specific *domain*. A working definition of a domain is an area of science or products where there is a common *ontology* Gruber:93].

Having a common ontology enables collaborators to work together with minimal risk of misunderstanding each other. When computer systems are used as the intermediaries in collaborative work, the need for a common ontology is even greater because many of the cues that exist in face-to-face interaction, the raised eyebrow, the wandering of attention, etc., cannot be perceived by one's partner.

The architectural aspect of a DSSA approach is that, once enough knowledge has been garnered about a specific domain, the object classes can be defined and placed in an operational relationship to each other [Haddock:94]. Within a domain, we assume consistency, namely, that the terms mean the same thing, i.e., refer to the identical object instances [Wiederhold:91], and have the same relationship.

3. DOMAIN DIFFERENCES

Having defined domains by their internal consistency, we must now consider the cases where such consistency does not hold. First of all, different domains will consider different objects. Different domains are likely to have different ontologies. These differences can be simply due to differences in naming and scope, both with respect to the names and semantics of meta-information about attributes that appear in the schema and the names and semantics that appear as values in the content of a database:

1 Naming attribute items differently. This is common, but also the easiest inconsistency to resolve. A example of this type occurs when employees are named in the `payroll` domain `EMP` and in the `personnel` domain `PEOPLE`. A simple table can be used to support the desired match and bring the information together.

2 Scope differences are much more insidious, and have to be determined by content analysis. The `personnel` domain my include assignees from other institutions, who are not listed in `payroll`. The `Payroll` may include support for student benefits for employee's children, but those children are, appropriatly, not listed in `personnel`. Resolution requires establishing, validating, and processing of rules. These rules can refer to variables in the domain that are not basic to the domain intersection.

3 Encoding differences of values are common as well. When numbers are used, a conversion can be established with a formula, say `meter = foot/0.305`. More complex are differences in dates and identifiers, say `ssn` with or without hyphens. Here rules have to be introduced as well, but when encodings are irregular, for instance `stock-codes`, tables have to be introduced. Tables dealing with instances of values occuring in databases require ongoing maintenance. We can hope that practical interoperation provides feedback which eventually will encourage coherence among domains.

4 Attribute scopes are often subjective. The term `hot` has a different meaning in the `weather` domain than the `truck-engine` or `truck-cargo` domains. If `hot weather` can effect the `truck-engine` expert knowledge is needed to make the

linkage. Differences in scope lead to differences in referencing, which makes their resultion yet more critical. For example, both `patients` and `nurses` are subsets of `people`, but their roles in a hospital are quite distinct, so that it would be unwise to create generalized `people` objects and encapsulate all the differences internally.

No central organization can resolve all these differences, they require knowledge about the source domains and their intersection.

The differences enumerated above can make a once-and-for-all integration of distinct domain infeasible. A dynamic capability is essential if we wish to achieve associative access to multiple domains, since the transformations required to achieve optimization must maintain the correct semantics. For instance, the PENGUIN system constructs objects as needed out of relational databases, given a structural model of connecting references [Barsalou:91].

4. DOMAIN MERGING

There are several approaches to dealing with building composed ontologies from domains that have ontological differences:

1 Aggregate the terms from all relevant ontologies, give them to a committee, and ask them to prepare definitions that are acceptable to all. When the definitions are completely documented, release the document and expect that all participants will adjust their usage to conform to the definitions.

2 Assume that terms match, and when mismatches are discovered, make the terms distinct, typically by prefixing them with source or domain identifier. This is the approach used by UMLS [Humphreys:92]; all the sources are labeled to make such distinctions easy, and by CYC, where micro theories can encapsulate differences [Lenat:90]. Over time, the processes of sharing of information encouraged by the availability of the joint ontology will cause convergence of meanings, although coherence can never be assured.

3 Assume that terms never mean the same thing unless explicitly instructed. Such instructions, encoded as *matching rules*, form a knowledge-base to be managed by collaborators from two or more domains. No restrictions are imposed on the evolution of local terms within a domain. Terms that are covered by matching rules form a new, second layer abstract ontology. Higher abstract layers can be defined recursively, leaving unneeded abstract terms local in their abstract layer.

We focus here on the third alternative.

5. An Example for Limited Domain Sharing

A multi-domain algebra needs the knowledge about the domains, specifically about the semantics of the intersecting terms.

We will illustrate the concept with a simple example:

S Domain S is of shoe stores, with objects as shoes to be sold, customers, their feet, sales people, business locations, and suppliers.

F Domain F is of shoe factories, with shoes being produced, lasts, materials as leather, glue, nails, and thread, suppliers for the material, employees, and production machinery.

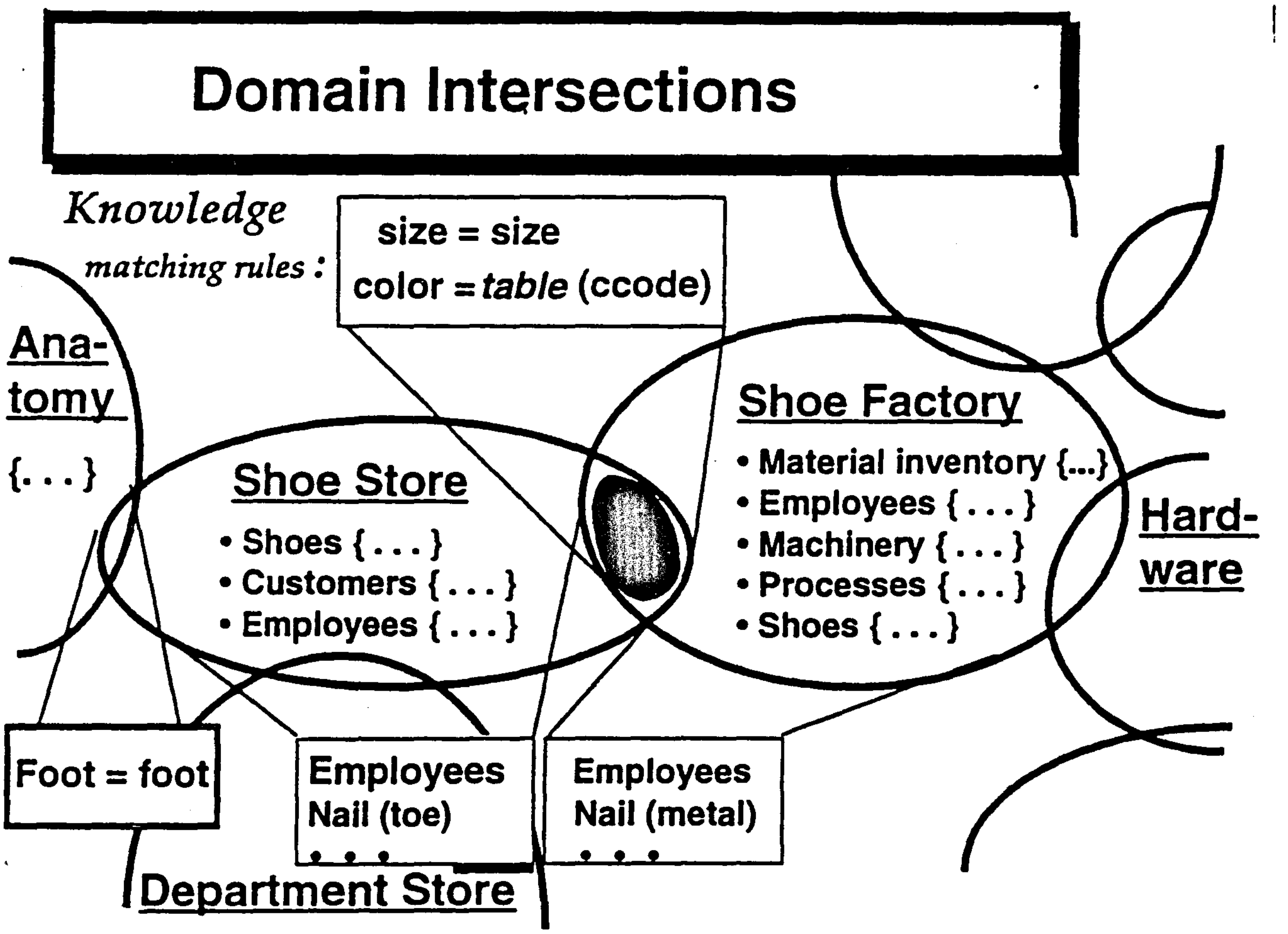
In order to create an information system that combines data from both, it is not necessary to merge both the S and F ontologies completely. Only terms along their connections must be merged, we assume by default that terms do not match. The required knowledge is:

```
S:supplier.name =  F:factory.name
S:shoe.size     =  F:shoe.size
S:shoe.color    =  color_table (F:shoe.color)
```

The color table provides the translation between the colors being attached to sales items, such as `pretty pink` and the color designation used in the factory, say, `13XF3`. Sometimes such relationships can be expressed as functions, say, conversions from `cm` to `inches`.

Not included in the knowledge-base, and hence not composable is the term `nail`, which in the store domain S is part of the customer's anatomy, and in the factory F designates part of the material used to make shoes. Similarly, the `employees` remain distinct, since the data collected for sales people differ from those in the factory.

The attached Figure illustrates the issues.



The income tax domain I will establish other connections between it and the sales and factory domains. A department store, incorporating many sales subdomains, will have more semantic connections, but still avoid needing an unconstrained union of all its ontologies.

We achieve scalability of information systems in this approach by the ontological partitioning [Gruber:94]. We enable composition over the parts by having a knowledge-based algebra. The individuals chartered with defining and maintaining the knowledge need more breadth than those that maintain domain-specific ontologies, but do not need the same depth of knowledge for the shoe supply connection. No knowledge about manufacturing detail is needed, although the factory may provide an abstraction called **quality**.

6. A DKB algebra

Given a formal Domain-Knowledge-Base model (*DKB*) containing matching rules that define sharable terms, the DKB-algebra should contain the following binary operations among domains:

Operation	symbol	semantics
DKB-Intersection	$\cap_{(DKB)}$	create a new subset ontology, comprised of sharable entries
DKB-Union	$\cup_{(DKB)}$	create a new joint ontology, labeling all but shared entries with their source
DKB-Difference	$-_{(DKB)}$	remove entries from an ontology, but shared entries are retained

Simple negation is avoided, so that no infinite ontologies are created.

Such an algebra can provide a basis for interrogating multiple databases which are sematically disjoint, but where a shared knowledge-base has been established. This process mirrors the approach used in CARNOT, where a knowledge base is used to create *articulation axioms* for joining of data [Collet:91]. However, CARNOT uses the default assumption that everything matches. When CARNOT uses a large and broad CYC knowledge base, many irrelevant retrievals can occur, so that in practice CARNOT system applications limit the depth of search.

An abstract layer created by taking the union ($\cup_{(DKB)}$) of several prior intersections ($\cap_{(DKB)}$) should not contain so many terms that coherence is hard to achieve. The relative autonomy of the local source terms provides scalability. The layered structure actually abdopts for information structuring the domain management strategy used by the INTERNET distributed naming conventions [Kahn:87].

With the conservative assumptions embedded in the *DKB-model*, the risk is that, because of having insufficiently many matching rules, too little information will be retrieved. By assigning the task of creating matching rules to many expert groups, we expect that high quality operations over data from distinct, but overlapping domains can be created at a reasonable cost. To evolve these systems effectively, feedback loops must exist that permit users to suggest new candidate matching rules, or to modify existing ones. Havingsmall, distributed groups to maintain the partitioned *DKB-models* will help ensure responsive maintenance of the domain knowledge.

7. CONCLUSION

Information technology is serving us well in specific domains, although we have remained dependent on specialist model designers and programmers for the implementation. Object technology lessens our dependence on specialists by being able to use an infrastructure which aggregates detail into meaningful units.

When the breadth of information system grows beyond coherent domains, further knowledge should be incorporated. To profit from such knowledge we propose a knowledge-based information algebra. The tasks of collecting and maintaining such algebras can be naturally partitioned among specialists and collaboratoring integrators. Integration can proceed at multiple levels of abstraction, avoiding the centralization that hinders progress in data exploitation of data from diverse sources.

Tools are needed to support such development, but to have effective tools a common formal structure is needed. Artificial Intelligence technology has been hard to scale when domains grew large or became diverse. The technology we describecan provide the needed formalism by building on relational algebras, formal management of semantics, and the incorporation of ontological concepts as a foundation for the management of the required knowledge bases.

REFERENCES

[Acton:93]C.H. Acton: "Using the SPICE System to Help Plan and Interpret Space Science Observations"; *Proceedings of the Second International Symposium on Ground Data Systems for Space Missions Operations*, Pasadena, California, November 16-20 (JPL Publication No. 93-5, March 1, 1993).

[Barsalou:91]T. Barsalou, N. Siambela, A. Keller, and G. Wiederhold: "Updating Relational Databases through Object-Based Views"; *ACM SIGMOD Conf. on the Management of Data 91*, Boulder CO, May 1991.

[Collet:91]C. Collet, M. Huhns, and W-M. Shen: "Resource Integrating Using a Large Knowledge Base in CARNOT"; *IEEE Computer*, Vol.24 No.12, Dec.1991.

[Dixon:69]Wilfrid J. Dixon and Frank J. Massey jr.: *Introduction to Statistical Analysis*; McGraw-Hill, 1969.

[Gruber:93]Thomas R. Gruber: "A Translation Approach to Portable Ontology Specifications"; *Knowledge Acquisition*, Vol.5 No. 2, pp.199-220, 1993

[Gruber:94]Thomas R. Gruber and Gregory Olsen: "An Ontology for engineering mathematics"; in Jon Doyle, Piero Torasso, and Erik Sandewall, Eds., *Fourth International Conference on Principles of Knowledge Representation and Reasoning*, Gustav Stresemann Institut, Bonn, Germany. Morgan Kaufmann Publishers, Inc., May 1994.

[Haddock:94]G. Haddock and K. Harbison: "From Scenarios to Domain Models: Processes and Representations"; *Proceedings of the Conference on Knowledge-based Artificial Intelligence Systems in Aerospace and Industry*, SPIE, April 1994.

[Humphreys:92]B.L. Humphreys and D.A.B. Lindberg: "The Unified Medical Language Project: A Distributed Experiment in Improving Access to Biomedical Information"; *MEDINFO 92*, North-Holland, 1992, pp.1496–1500.

[Kahn:87]Robert E. Kahn: "Networks for Advanced Computing"; *Scientific American*, Vol 257 No.5; Oct.1987, pp.136-143.

[Lenat:90]D. Lenat, R.V. Guha, et al.: "CYC: towards programs with common sense"; *Communications of the ACM*, Vol.33 No.8, Aug.1990.

[Loomis:87]Mary E.S. Loomis: *The Database Book*; MacMillan, 1987.

[Rose:83]Robert F. Rose: "A 'Data Engine' Using SAS and INQUIRE"; *Journal of Medical Systems*, Vol.7 No.3, 1983, pp.257–266.

[Wiederhold:91]Gio Wiederhold: "The Roles of Artificial Intelligence in Information Systems"; *Journal of Intelligent Information Systems*, Vol.1 No.1, 1992, pp.35–56.

[Wiederhold:92]Gio Wiederhold, Peter Wegner, and Stefano Ceri: "Towards Megaprogramming"; *Comm. ACM*, November 1992, pp.89-99.

Making Specification Design More Accountable

Dave Robertson, Jane Hesketh
Department of Artificial Intelligence,
University of Edinburgh,
80 South Bridge, Edinburgh
Email: `dr@uk.ac.ed.aisb`, `jane@uk.ac.ed.aisb`

Abstract

If something goes wrong with a software/hardware implementation then it is important to know what caused the problem. Sometimes the error lies in incorrect implementation of a specification but it is thought that a more common source of problems lies in earlier stages of the development process, where there may be mismatches between the requirements of the domain and the specification. For this reason, it is common (particularly in safety-critical applications) to find tightly monitored regimes of specification, in which the strategies of specification designers follow prescribed conventions and are closely constrained by regulatory reviewers. It would be useful if this process were as explicit as possible, making the design and reviewing process more open and accountable. One way of doing this is to provide formal descriptions of design strategies and to require designers to endorse their use of these by reference to appropriately formalised parts of the regulations. We examine how this may be done, using as a concrete example the domain of oil platform emergency shutdown systems.

1 Introduction

Our general approach is to follow an existing methodological framework, using formal methods to reinforce key parts of existing practice. Figure 1 gives a sketch f the main components of this framework and their interactions. Our prime concern is to provide an acceptable system specification, which we would expect implementation designers to realise in appropriate software and/or hardware. To be acceptable, the specification must address more than just the behavioural requirements stipulated for the domain of application. It must also be considered "well designed", by conforming to standard practice for the the specification language concerned. Furthermore, it must have been "carefully designed" - that is, designers should be able to show that they have taken appropriate codes of practice for the domain into account when constructing the specification. Ideally, they should be able to annotate appropriate parts of the specification with evidence to show that they have paid attention to standard design principles and codes of practice. This is what we mean, in this paper, by accountability in specification design.

To ground our discussion in a realistic domain, we consider the use of this methodology in specifying oil platform emergency shutdown systems. Oil production platforms are complex and expensive systems in which it is necessary to place humans in close proximity to potentially hazardous industrial processes. This situation is sometimes caricatured as "building a hotel next to a chemical plant". It is therefore essential that the emergency shutdown systems on production platforms function reliably and that the design of such systems is performed in a highly disciplined and accountable manner. We shall begin our discussion by presenting an account of the specification methodology of this domain (following the template given in Figure 1). We next describe a mechanism for formalising part of standard design practice (using schematic design components) and then show how this can provide a framework for attaching endorsements on design decisions in terms of parts of a code of practice. We conclude by raising some questions which we believe are particularly germane to this line of research.

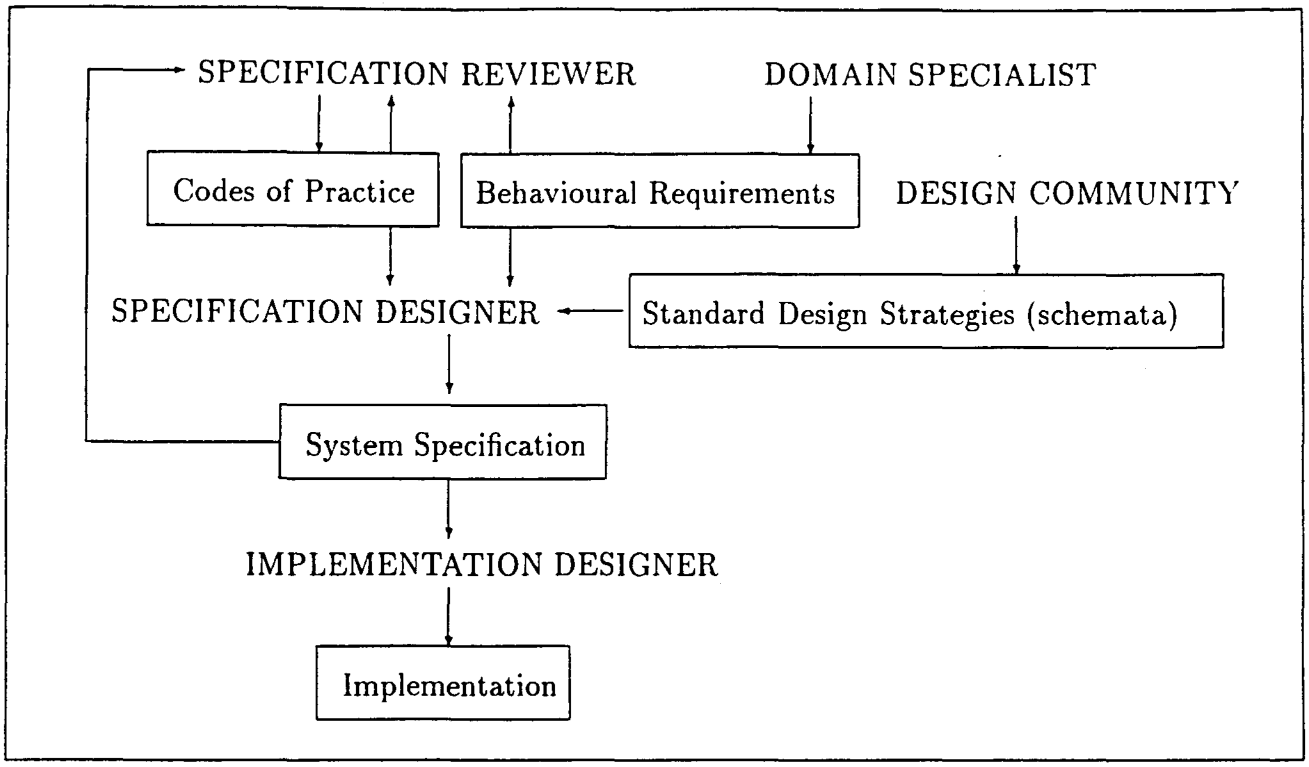
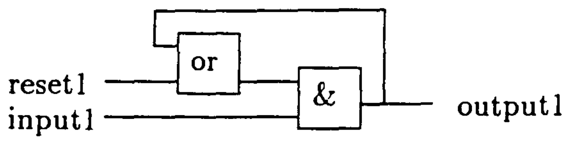


Figure 1: Schematic diagram of one type of specification process

2 Sketch of the Shutdown System Design Process

The entire process of production platform design and commissioning is highly complex – involving collaboration between government, oil companies and numerous subcontractors. We shall concentrate on the core of this process, which involves the requirements stipulated in an oil company's code of practice and its interaction with the specifications used as the starting point for system design. Because of the high risks involved, the company provides a detailed code of practice for emergency shutdown and process trip systems. This contains a mixture of information, ranging from general goals which must be achieved by the system engineer (*e.g.* that all the relations stipulated in the cause–effect matrix should be accounted for) to specific requirements for system behaviour (*e.g.* that trips should latch outputs in a de–energised state). The design of a shutdown system begins by determining the inputs to the safety system (from devices such as sensors or trip switches) and the outputs (which activate alarms, shut down machines, etc). A typical safety system might have 1500 inputs and roughly the same number of outputs. To provide a coarse–grain description of the relationships between inputs and outputs, a cause–effect matrix is drawn up. This gives, for every input cause, the output effects which should be initiated if the input is tripped. Tripping is normally signalled by de–energising, thus ensuring that power failure initiates safety shutdown. Given a cause–effect matrix, the next step is to design a specification for the safety system such that the stipulated cause–effect relations are obtained and that conforms to the company code of practice requirements. The standard language for specification is "functional logic", which is a diagrammatic language based on standard logic circuit elements with time delay components. There is a high degree of standardisation in the construction of these specifications, with similar configurations frequently being re–used.

Strong links between requirements and specification are important at two stages in the evolution of the specification. The most obvious of these is in representing standard patterns of design and in



```
value(output1,Time,Value) ←                       % A Value for output1 at Time is obtained if
        previous_time(Time,Tp) &                  % Tp is a the time point before the given Time and
        value(output1,Tp,V0) &                    % V0 is a value for output1 at Tp and
        value(reset1,Tp,V1) &                     % V1 is a value for reset1 at Tp and
        value(input1,Tp,V2) &                     % V2 is a value for input1 at Tp and
        or(V0,V1,Vor) &                           % Vor is the logical 'or' of V0 and V1
        and(Vor,V2,Value)                         % Value is the logical 'and' of Vor and V2.
value(output1,Time,Value) ←                       % A Value for output1 at Time is obtained if
        initial_value(output1,Time,Value)         % Value is the given initial value for Output, with
                                                  % Time being the initial time.
```

Figure 2: Reset configuration: diagram and FOPC interpretation

linking these to the appropriate details of the code of practice during program construction. This provides a formal framework for specification, making the connection between design decisions and company requirements more explicit. A less obvious, but perhaps more important, role is during later maintenance of the system. If part of the implemented system needs to be changed then it is important to ensure that this alteration remains consistent with the code of practice. However, the connection to the code of practice from the specification diagrams may not be obvious, unless the original designer has kept detailed records of the links between the two. In the next section we shall describe the techniques which we have employed to maintain these types of links.

3 Specification Schemata

In Section 2 we noted that designers frequently re–use variants on standard configurations of specification components. We refer to such configurations as schemata. One of the simplest examples of a schema is the reset configuration, an instance of which is shown in Figure 2. The diagram shows the standard method of displaying this configuration with an annotated Horn clause interpretation of this diagram appearing below it[1]. A reset configuration is used to allow a trip to be reactivated after deactivation. Thus, in the circuit of Figure 2, if *input1* is deactivated then *output1* should be deactivated and remain so until *reset1* is activated at the same time as *input1*. Note that the Horn clause interpretation we have given for the functional logic is not necessarily the "correct" one, since this depends on the level of detail with which one wishes to model the behaviour of the circuit. We shall return, briefly, to this issue in Section 5.

If we examine the circuit in Figure 2, we can see that it contains elements which distinguish it as a reset component: a feedback loop from the output to a logical 'or' with the reset, with the resulting value being 'and'ed with the input. However, there is flexibility in this definition - in particular, we could cater for a conjunction of more than one input signal simply by installing a sequence of 'and' connectors (one for each additional input, with the output from each 'and' being carried forward to

[1]We adopt the Prolog convention that constants are represented with words beginning in lower case letters, whilst variables begin with upper case letters and are implicitly universally quantified

```
schema(reset, Agenda, Output,
        { value(Output, Time, Value) ←
                    previous_time(Time, Tp) &
                    value(Output, Tp, V0) &
                    value(Reset, Tp, V1) &
                    value(Input, Tp, V2) &
                    or(V0, V1, Vor) &
                    AndGoals,
          value(Output, Time, Value) ←
                    initial_value(Output, Time, Value)},
        reset_cluster(Reset, Inputs),
        generate_and_sequence(Inputs, Tp, Vor, Value, AndGoals),
        {Reset} ∪ Agenda).
```

Figure 3: Reset configuration schema

the next). Thus, we can define a general reset schema which, when given the appropriate subset of inputs which need to be allocated a reset, will generate the necessary circuit specification. In general, our schemata have the form:
schema(Name, Agenda, Output, Axioms, Problem, Constructor, Agenda′)
where: *Name* is the name of the schema; *Agenda* is the set of outputs which need to be accounted for in the design at the time of applying the schema; *Output* is the output which the schema produces; *Axioms* is the set of axioms necessary to produce the output; *Problem* is the design requirement which needs to be satisfied in order to use the schema; *Constructor* creates the necessary axiom structure; and *Agenda′* is the set of outputs which need to be accounted for after applying the schema. An example of a general schema for the reset assembly described above appears in Figure 3. The design requirement for this schema (*reset_cluster(Reset, Inputs)*) is that in the requirements there should be some combination of *Inputs* which have to be connected with the given *Reset* switch. The *generate_and_sequence* constructor produces the appropriate sequence of 'and' connectors, thus instantiating the variable *AndGoals*.

4 Constrained Generation from Schemata

The schemata of Section 3 can be harnessed to provide a goal–directed mechanism for generating functional logic specifications, starting from the outputs of the cause–effect matrix and working "backwards" to account for each of these in terms of the inputs. The essence of the agenda–based generation algorithm is described in Figure 4. Note that the search space for this algorithm is potentially infinite, since the application of a schema can increase the size of the agenda of outputs. The key to the successful operation of this algorithm is to be highly selective in the choice of schema for each output in the agenda, so that the specification is not only as compact as possible but also remains faithful to the requirements of the code of practice.

In Figure 3, we have already given an example of a requirement at the level of the design itself which helps to constrain the selection of appropriate schemata. This was the schema problem condition: *reset_cluster(Reset, Inputs)*, which collects the necessary reset and input identifiers for a reset configuration. We expect the decision about which these should be to be taken by the specification designer (assisted as far as possible by our system). However, the designer must also indicate why

Assume we are given a cause–effect matrix with set, $\mathcal{I}$, of inputs and set, $\mathcal{O}$, of outputs. The predicate *generate*($\mathcal{I}, \mathcal{O}, \{\}, \mathcal{A}_F$) will, if successful, produce a set $\mathcal{A}_F$, of axioms describing the functional logic specification linking $\mathcal{I}$ to $\mathcal{O}$. This is defined recursively as follows:
generate($\mathcal{I}, \mathcal{O}, \mathcal{A}, \mathcal{A}_F$) when:

- If $\mathcal{O} = \{\}$ then $\mathcal{A}_{\mathcal{F}} = \mathcal{A}$.
- Otherwise, remove an element, O_1, from $\mathcal{O}$ leaving remaining elements $\mathcal{O}_R$ and:
  - If O_1 is already accounted for in $\mathcal{A}$ then *generate*($\mathcal{I}, \mathcal{O}_R, \mathcal{A}, \mathcal{A}_F$).
  - Otherwise, select a schema such that:
    - *schema*($Name, \mathcal{I}, \mathcal{O}_R, Axioms, Problem, Constructor, \mathcal{O}_N$)
    - and the conditions in *Problem* can be confirmed from user requirements.
    - and *Constructor* can generate the necessary structures in *Axioms*.
    - $\mathcal{A}_N$ is the result of merging *Axioms* with $\mathcal{A}$.
    - and *generate*($\mathcal{I}, \mathcal{O}_N, \mathcal{A}_N, \mathcal{A}_F$).

Figure 4: Simplified schema application algorithm

this choice was made by referring to the code of practice, otherwise it will be difficult to see how these more general requirements influenced the design. An example of a relevant general requirement from the code of practice is:

> "Attempted Operation of any reset facility with a trip demand still active shall have no effect."

This requirement does not, directly, determine the details of the specification design – it imposes a condition on the behaviour of the completed system. Thus, the choice of a particular instantiation of the reset schema of Figure 3 might be endorsed by appealing to the above excerpt from the code of practice. By forcing designers to make their assumptions explicit in this way, it becomes easier for other experts to assess whether they are warranted.

Where requirements in the code of practice are sufficiently detailed, we may provide formal interpretations of these. However, since sentences in natural language typically admit a variety of interpretations the way in which we formalise requirements at this level may not be uniquely determined. In other words, there may be more than one way of demonstrating that a requirement holds. Taking the reset requirement above as an example, one interpretation of this might be that if there is an initiating signal to one of the trip devices then there should be no combination of inputs to the other devices which can reactivate one of the outputs which it was designed to trip. Figure 5 gives one way of formalising this requirement. The axiom in this figure says that if I is a trip device and A is an action which should follow from I (as indicated by the cause–effect matrix and the initiatin signal for I is V_I and $\mathcal{S}$ is the set of all inputs other than I and the activation signal for A is V_A, then it should not be possible to find an time later than some initial time point, T, such that an assignment of activation values can be made to $\mathcal{S}$ and, using these, the activation value V_A is derived for A. This more detailed rendering of the natural language requirement could be used by the designer to stipulate the type of verification which he/she recommends as a way of confirming that the requirement has been met.

Given the functional logic axiom set, $\mathcal{A}$:

$$\begin{aligned}
&\mathit{trip\_device}(I)\ \&\ \mathit{action\_of}(I,A)\ \&\\
&\mathit{initiation\_signal}(I,V_I)\ \&\ \mathit{activation\_signal}(A,V_A)\ \&\\
&\mathcal{S} = \{X \mid (\mathit{input}(X)\ \&\ \neg X = I)\}\ \&\\
&\mathit{initial\_time}(T) \quad \rightarrow \quad \neg(\ \mathit{assign\_values}(\mathcal{S},T,\mathcal{V})\ \&\\
&\qquad\qquad\qquad\qquad \mathit{later\_time}(T,T_F)\ \&\\
&\qquad\qquad\qquad\qquad \{\mathit{input\_value}(I,\mathit{AnyT},V_I)\} \cup \mathcal{V} \cup \mathcal{A}\ \vdash\ \mathit{output\_value}(A,T_F,V_A))
\end{aligned}$$

Figure 5: Requirement endorsement

5 Conclusions

We have described, in simplified form, a system for constructing specifications of shutdown systems using schemata to capture standard design strategies for portions of the system, with a goal-directed specification generator tying the portions together. Our generation algorithm if unconstrained, would be capable of generating a huge variety of possible configurations so there needs to be some way of constraining the search. This is done by associating preconditions with each schema, determining its applicability to the problem which the designer is tackling. Designers must also endorse their choice of schemata by appealing to appropriate excerpts from the code of practice. Some of these excerpts may be interpreted formally – thus allowing designers to stipulate what they consider to be appropriate tests of compliance with the requirements.

The domain chosen for our experiments is, we feel, particularly appropriate for the use of formal methods. The high stakes involved in oil production, plus the strong emphasis on accountability and monitoring, mean that the effort of producing a formal framework for interpreting requirements may be justified. However, we have applied a similar approach, with some success, in the domain of ecological modelling [2]. In the ecological domain, the emphasis was on the use of requirements as a way of buffering users from the details of specifications of ecological models. By contrast, in the domain of safety shutdown systems we expect users to be highly conversant with specification details, with requirements being used primarily as a means of endorsing design decisions.

The strongest limitation on our system is the range of design strategies encompassed by the library of schemata which we provide to designers. Our approach works best when there is a high degree of standardisation in specification style. If users wish to stray beyond these limits then we must provide them with general purpose schemata which either require a high degree of parameterisation by the user or individually contribute only small portions of the design. One way of achieving generalisation in a more disciplined way, in the context of Horn-clause specifications is to use notions of specification "techniques", enabling incremental construction of predicate definitions. A summary of recent Edinburgh work in this area is given in [1].

A further, fundamental, issue in the formalisation of requirements in systems such as ours is the degree of expressive power which we allow users in describing their problems. Highly expressive languages, although perhaps theoretically appropriate, may not easily be understood by users in the domain. Simpler languages may be easier to present to users but may provide insufficient information to control the generation of specifications. In [3] we have examined this issue in the context of our work in the ecological modelling domain.

References

[1] A. Bowles, D. Robertson, W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying programming techniques. *International Journal of Man-Machine Studies*, in press.

[2] 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.

[3] D. Robertson, M. Uschold, A. Bundy, and R. Muetzelfeldt. The ECO program construction system: Ways of increasing its representational power and their effects on the user interface. *International Journal of Man Machine Studies*, 31:1–26, 1988.

Formal Methods for Complex Evolving Systems

July 29, 1994

Steve Vestal
Honeywell Technology Center
Minneapolis, MN 55418
(612) 951-7049
vestal@src.honeywell.com

Introduction

We take a reasonably broad view of what a formal method is and how it can be usefully applied. A formal method is any body of theory of a mathematical nature that can be applied to some aspect of a software product. A formal method can be of value because it provides a sound basis for specification, or a sound basis for design, or a sound basis for analysis and estimation of product characteristics. Any particular formal method allows users to analyze or verify particular properties or characteristics of a software product, where different properties or characteristics are addressed by different formal methods. For example, real-time schedulability theory and formal language theory are both formal methods, where each can be used to guide design in different ways and verify different kinds of properties. Important defining characteristics of formal methods are that they are mathematical in nature and provide some sort of sound analysis or verification or reasoning methods.

Software should not be viewed as a monolithic and independent technology or industry. Software should be viewed as a medium of expression, where one must pay careful attention to the ideas and technologies being expressed in software in a particular market or application domain. For example, the skills, technologies and processes used to develop avionics or industrial process control software have some significant differences from those used to develop accounting systems or compilers. Different combinations of formal methods are needed to address the needs of different application domains.

We propose two important challenges in the development and application of formal methods: the identification of new and continued development of known formal methods that can be applied in application domains of importance; and the integration of multiple formal methods to more comprehensively address the properties and characteristics of products in particular application domains. Such activity is enabled by emerging theoretical work that increasingly addresses real-world details, emerging approaches to the application of formal methods that mitigate many previous concerns, increasing industrial openness to formal methods, and increasing market acknowledgement of the needs that can be addressed by formal methods.

This paper is based on an approach we have been taking in the ARPA/ONR Domain-Specific Software Architectures program to integrate, apply and transition selected formal methods into embedded software application domains. Our approach is based on identifying widely-usable idioms for computation, communication and control at the system or architectural level; developing formal notations based on these idioms that allow users to describe

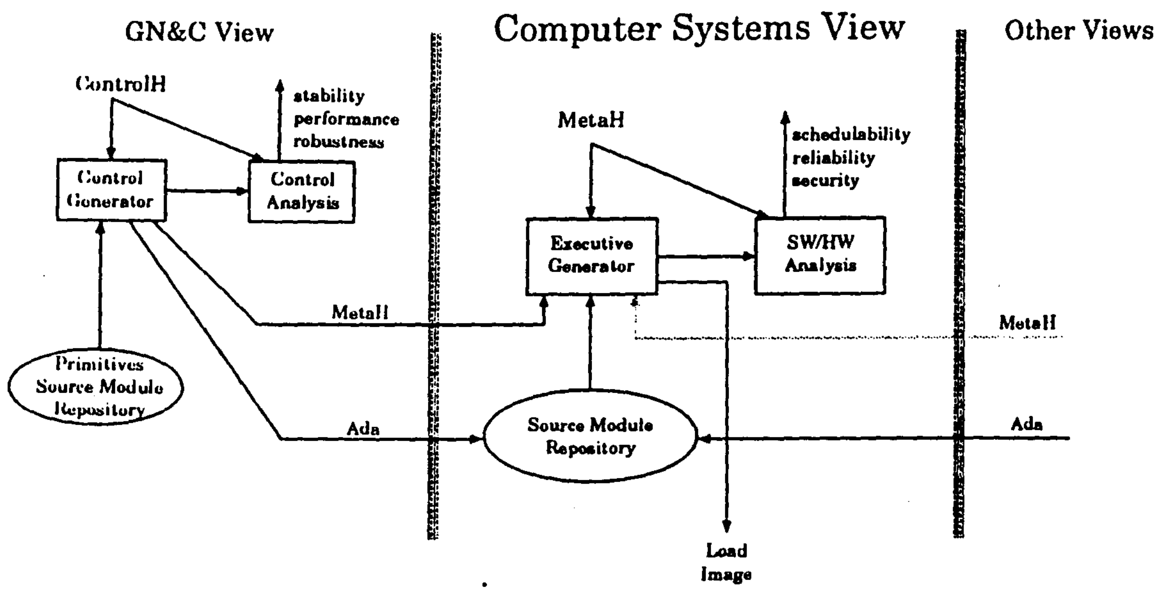


Figure 1: DSSA for GN&C Architecture Description Languages and Tools

how a system is built by composing source modules; and using selected formal methods to provide semantics for these architectural notations and to analyze and verify specified architectures.

A key aspect of this approach is that by codifying the way that formal methods are integrated and used within an application domain, and by automatically co-generating an implementation together with various formal models from the same architectural specification, we mitigate objections to the use of formal methods as "too complex" or "too time-consuming," at least within application domains for which solutions can be specified using our architectural notations. From a business perspective, this approach can provide a high return on the investment made to obtain the various development assets (which include the various formal methods, models, tools and techniques).

We will first try to make this approach somewhat more concrete by describing as an example the architectural notations, tools, and design and implementation techniques we are developing. In keeping with the emphasis of the 1994 workshop, we will then briefly discuss how our approach addresses two important practical problems: first, how can multiple specifications and associated formal models and other assets be used together in a way that supports multi-disciplinary evolutionary development and verification; and second, how can on-line upgrades to a executing system be made more quickly and reliably.

DSSA for GN&C

Figure 1 shows at a high level the architectural notations (called architecture description languages) and tools that we are developing.

ControlH is a language used by guidance, navigation and control (GN&C) engineers to specify models of external objects (e.g. aircrafts, sensors, actuators) and algorithms used to control them. ControlH is based on the block diagram notation common in the field and uses differential and difference equations as semantic models. The language also includes a number of advanced features such as a rich set of types, type inferencing, and generic operators to support reusable GN&C models and designs. In addition to automatic code composition/generation, tools are available to provide simulation, equilibria determination and linearization, and (from several commercial vendors) to perform various standard mathematical analyses.

MetaH is a language used by computer systems engineers to specify how systems are composed from such objects as subprograms and packages, processes, message and event connections, shared data, modes of operation, processors, memories, and inter-processor communication hardware. MetaH can be used to hierarchically combine simpler components to form more complex "mega-components," and to combine mega-components (possibly generated by various discipline-specific toolsets like ControlH) to form an overall product. MetaH semantics are based in part on fixed priority preemptive scheduling theory, and we are currently working on stochastic models for certain classes of events (e.g. aperiodics, fault events). In addition to automatic code generation/composition, tools are available to support software/hardware binding and perform real-time schedulability analysis. Future extensions are tools for reliability analysis, secure partitioning analysis, and stochastic performance analysis.

The overall effort represents a selection, integration and automation of analysis, design and implementation technologies that enable us to specify reusable software architectures for a reasonably broad class of embedded software products. Associated with these architecture description languages are tools to perform various analyses based on the formal models used to provide architecture semantics, and tools to generate/compose source modules to provide an implementation for a sufficiently detailed architecture specification. Important aspects of these languages and tools are support for partial and evolving specification and analysis; the use of formal methods to provide analysis for design trade-offs; co-generation of implementation and formal models to increase assurance of implementation correctness; and well-defined mappings between specifications and well-structured outputs to support inter-disciplinary design trade-offs and to support system-level verification.

Composing Formal Methods and Architectures

Some important challenges in developing our toolset have been deciding where and how to decompose overall system specification into different languages that provide complementary views of a system; and the selection and integration of formal methods to provide reasonably broad semantics with automated, consistent analysis and implementation.

Part of our solution has been to provide two different specification languages to provide two different views of a system for two different disciplines: ControlH for GN&C engineers and MetaH for computer system engineers. Other views are possible (e.g. we have looked into display management and signal processing views). Important factors are the degree to which the views are "orthogonal" in terms of separability of concerns during specification and analysis; and the ease of composing source modules produced in the different views.

Most of the primary performance and robustness concerns of the control engineer, explored in part by analyses such as pole plots and μ analysis, are indirectly related to issues such as real-time schedulability and software/hardware partitioning. There are a few basic constructs and concepts common to both languages, such as time, periodic processing, and operators/subprograms.

The mega-components generated by the ControlH toolset, when viewed as software entities by the computer systems engineer, form a collection of subprograms, processes and connections that plug fairly naturally into an overall MetaH architecture. The external interfaces and real-time requirements from the control engineer's perspective can be captured in MetaH, and in fact our ControlH toolset also automatically generates MetaH specifications that describe the generated GN&C source modules.

An important aspect of all this, in our opinion, is using code assembly/generation and model generation technologies that produce well-structured outputs having a reasonably intuitive mapping back to the original specification. The resulting traceability extends across the interface between the different views, to the extent that a computer systems and a control systems engineer can discuss the results of the various analyses and make multi-disciplinary design trade-offs together, even though each has only a limited understanding of the impacts and alternatives in the other's field.

Within each view, multiple formal methods need to be used in an integrated way. For example, within our MetaH view, fixed priority preemptive scheduling theory is used to provide temporal semantics for specifications and to analyze the temporal behavior of the implementation. Concurrent state machine models are used to specify the configurable executive that implements the real-time process model seen by users, which leads to increased assurance of the soundness and completeness of the design. Within each view there is a collection of mutually consistent source code and formal model artifacts, written in particular languages for particular tools. This approach enables multiple, consistent analysis and verification capabilities within each view (for more details see Binns, "Real-Time Software and Proof Architectures," in these proceedings.)

Reliable Upgrades to Executing Systems

Some applications need to be modified on-line (e.g. industrial process control systems, telecommunications systems). The justifiable hesitance to make such upgrades without extensive testing and analysis has been identified as a major impediment to upgrading systems with newer technology and evolving towards more "open" systems. Trust is a human quality, and a major practical contributor to trust in a system is a long-term history of correct operation.

Our work extends the degree to which various methods can be applied to assure the correctness of a product prior to fielding that product, including assuring the correctness of a modification to a product. However, there always comes the "moment of truth" at which the on-line system is modified during operation. We will outline two enabling technologies that could help make it easier to face the moment of truth: secure partitioning and dynamic mode changes.

Secure partitioning is a concept recently introduced into commercial avionics systems. Different system components have different criticalities assigned (e.g. navigation is more critical than management of passenger entertainment facilities). A kernel that provides secure partitioning enforces constraints at run-time that assure the consequences of software defects cannot propagate from lower criticality components into higher criticality components (this is an example of a security policy that might be verified by a MetaH security analysis tool). Secure partitioning requires traditional address space protection and capability checking as well as enforcement of real-time temporal constraints (all of which MetaH supports). Assuring correctness at high criticality levels is extremely expensive and dominates the development cost of such components. Secure partitioning allows this to be done only for that relatively small fraction of the application that is genuinely critical. In addition to cost and time savings, this allows COTS software (whose correctness usually cannot be assured to any significant level of criticality) to be used for low criticality functions. The enabling technologies are methods for developing and assuring the correctness of the enforced secure partitioning constraints and kernel to the highest level of criticality.

In MetaH, a mode of operation is a collection of processes together with associated event and message connections that are active simultaneously throughout some interval at run-time. Multiple modes may be specified, with event connections defining the pattern of mode changes that can occur at run-time. A mode change may stop some processes, may start some processes, may change some event connections, or may change some message connections. It is important to note that a mode change may affect only a portion of an executing system. An important implementation detail is that a mode change protocol is used that provides synchronized fault-tolerant real-time mode changes in multi-processor systems.

Together, such features could be used as follows to increased assurance that an on-line change to an executing system cannot lead to a catastrophic failure (along the lines of the CMU/SEI recovery/renewal work). Each change is first specified as a delta to the specification of the modes of operation of the currently executing system. That is, each change is structured as an added set of modes. A design principle followed in all versions (old and new) is that there are always trusted versions of critical functions and trusted monitors capable of determining whether other implementations of those functions are operating acceptably. Whenever any monitor detects an anomaly, it triggers a change to a mode in which the trusted function assumes control. Highly trusted functions can often be very simple (and thus easy to verify), the primary difference between a trusted and a new component being the level of performance or optimality achieved.

All desired off-line verification and validation methods would be employed. The new modes would then be loaded into the executing system in background. The moment of truth consists of a mode change from the current to the new mode of operation. Afterwards, if a trusted monitor detects any anomalies then a change back to a trusted mode of operation is triggered by that monitor. Any defect-induced errors in a new component are prevented from propagating by secure partitioning. As a system evolves, antiquated modes of operation would be cleaned out.

An annotated bibliography and copies of several papers can be obtained via anonymous ftp from src.honeywell.com, pub/dssa/papers.

Formal Methods Technology Transfer: Impediments and Innovation

Dan Craigen and Ted Ralston
Odyssey Research Associates (ORA)
email: dan@ora.on.ca and ted@oracorp.com

September 19, 1994

1 Introduction

The following is a sober but constructive view on the current state of formal methods. The purpose of expressing this perspective is to start the lengthy process of rehabilitating formal methods by facing our problems directly. Hopefully, by being realists, formal methodists will be able to improve upon the current state of affairs and meet the needs of the defense, and more broadly, commercial communities. The paper concludes by analyzing data from a survey of industrial applications of formal methods[1] [11] from the perspective of a particular innovation adoption model. Such an analysis can provide recommendations of how best to overcome the technology transfer problems facing formal methods.

2 Failure of Technology Transfer

Today, the main issue facing formal methodists is the technology transfer of a formal methods capability from the realm of formal methodists to the broader communities of defense and commercial system developers. Even though there have been a small number of successful applications of formal methods technology, in which improvements in quality and reductions of cost and time-to-market have been reported [11], there is a clear resounding failure of technology transfer. While technically advanced systems (such as ORA's EVES [1, 2] and Penelope [3] systems, Computational Logic's Boyer-Moore-based systems [4, 5], SRI's EHDM [6] and PVS [7] systems, to name only a few) are successful technical developments, their effect on the defense and commercial communities has

[1] In addition to the two authors of this position paper, Susan Gerhart (now at the University of Houston at Clear-Lake) was a principal in performing the survey.

been minimal.[2] We must recognize that technology transfer is not solely an issue of developing an eminent technology: there are also active political, sociological and economic forces that act as impediments. Consequently, we can only conclude that fundamental change in the development and marketing of formal methods are necessary to overcome these impediments to diffusion.

3 Requirements for New Technologies

In his thesis [8], Nico Plat expresses an opinion that:

> "New Technologies are not only required to be effective, but also have to solve a perceived problem and have to be acceptable to their intended users."

While there are undoubtedly other requirements for new technologies, for the purpose of this paper and in the remainder of this section, we will look at the three properties that Plat feels are fundamental:

- solution to a perceived problem,
- effectiveness of formal methods technology, and
- acceptability of formal methods to their intended users.

3.1 Perception Problem: We don't have a perceived problem!

Even with the ongoing gripes about the "software crisis,"[3] industry and the public do not yet believe a problem truly exists in software or systems development.[4] Substantial fortunes are being made by computing technologists and marketers and many systems appear to work satisfactorily, if not with a high degree of reliability. With the insertion into society of tools that can replace intellectual effort (e.g., the use of calculators to perform simple arithmetic operations), concerns have been voiced about the "dumbing down of society." Coincidentally, consumer expectations of software appear to be substantially below what is expected of other commercial or defense products.

[2] It is, perhaps, of some note that two European based tools and methods ([10, 9]), appear to have much broader user communities than the union of the communities of the aforementioned systems.

[3] More accurately a "software condition" or "software fact of life" as a crisis does not last for 25 years. A crisis is, according to The Concise English Dictionary, a momentous juncture in war, politics, commerce, domestic affairs, etc.

[4] In part, this belief appears to be a result of the generally perceived improvements in productivity arising from the use of software products, even though those products may manifest surprising behavior from time-to-time.

Unfortunately, we have come to the view that only a series of catastrophes, directly attributable to poor software, will result in any change in perspective and consequent expectations.[5] So, even though a survey by Genuchten showed that for 191 organizations, the average cost overrun of projects was 33% and that projects were sometimes or usually late in nearly 80% of the organizations surveyed, the status quo persists.

Another indication of the perceived wisdom that a "software condition," does not exist, is the ongoing growth in the complexity of systems envisioned and implementations attempted. Software Engineers and others involved with the development of systems, have yet to accept or even define what are the engineering limitations of the technology. From a historic perspective, software is not unique in pushing the limits of extant technology. Much of engineering has its roots in empiricism, in which artifacts were built, (based on the folklore, craft and, yes, science of the day), and failed when limits were exceeded. Bridges and buildings collapsed and planes crashed. Once the science and engineering of the physical realities were understood, artifacts were built within the relevant constraints: Our discipline, software engineering, has not reached that level of maturity.

A rather topical example is the current problems with the baggage handling system at the new Denver airport. According to a New York Times article, cost overruns are in the order of $1 million per day and an expectation that the total cost of the airport will double to nearly $2.5 billion. Recently, the press has reported that the airport authority has decided to build a second, conventional, baggage handling capability.[6] The failed developments of other complex systems, such as the British-based attempts to develop an automated London ambulance dispatch system and a software based stock trading system, along with the whole litany of failures reported in Peter Neumann's Risks Forum, accentuate lack of discipline.

3.2 Are formal methods effective?

If one takes a cold scientific perspective on this question, the best one can respond is that the case for formal methods is inconclusive. Various surveys (e.g., [11]) have provided reasonably systematic anecdotal evidence of effective industrial use of formal methods. However, none of the formal methods application projects have followed strict scientific principles and the putative benefits may arise for other reasons–excellent staff or an improvement in process, for example.

[5] Admittedly, catastrophe is a strong word to use. It may be that a series of failures (e.g., consider the current public relations image of NASA) or security breaches (e.g., the penetration of the tax and/or health records of public figures) will suffice.

[6] Not being familiar with the technical difficulties being encountered with the baggage handling system, we do not make any claims as to whether the application of formal methods would or could alleviate the situation. One suspects the problems are broader than purely technically oriented.

So, the claims of formal methodists are primarily based on religion and anecdote. (Mind you, formal methodists are not unique in making such claims.) This is not to say that the claims are necessarily false; only that they are scientifically unproven.

Additional problems arise because of the lack of metrics for estimating the cost of projects that use formal methods. It appears that formal methods projects skew the development curve so that significantly more effort is directed at the requirements and specification stages. This results in edgy managers who are used to seeing code being produced rapidly.[7] Hence, there is a perceived risk to ameliorate.

The potential effectiveness of formal methods has further been harmed by the profusion of unreliable, cost ineffective and weak tools. Tools are meant to extend human capabilities. By using cars and planes, one travels further; by using telescopes and spectacles, one sees further. Formal methods tools should perform similar feats but, on the most part, do not, as they are still labour intensive and the labour involved is usually highly skilled.

3.3 Are formal methods acceptable to their intended users?

In general, we must conclude that the answer is a resounding *no*. There is a huge chasm between formal methods experts and traditional software engineers and mangers. While there is resistance arising from inertia, resistance also arises because of different world views. Logical notation, proof systems, the methods of formal methods are all quite alien to software engineers and managers. Even the idea of disciplined development is anathema to many, so called, professionals. Our survey [11] discusses a number of barriers to the successful transition of formal methods technology.

4 Adoption of Innovations

Having painted a rather sobering picture, we now aim to be constructive by analyzing the data provided by our international survey of industrial applications of formal methods [11] from the perspective of adoptability in order to assess the problems so far encountered and suggest possible solutions. The current analysis (which is still rather preliminary) differs from the effort in [11], in that the survey analysis was a "common sense" analytic framework.

An analytic framework used in this analysis is borrowed from the study of innovation carried out in the 1970s and 1980s by several scholars and industry associations (e.g., E.M Rogers, Nathan Rosenberg, Richard Collins, The Sloan School of Management, the American Electronics Association, and the U.S.

[7] Of course, that formal methods have led to increased effort at the requirements stage is beneficial; studies have shown that around 60% of system errors can be attributed to poor requirements.

National Academy of Sciences, to name a few). This framework has come to be accepted by both academia and industry as a useful way to assess strategies for innovation adoption and has been useful for identifying impediments.

In this analytic framework, the criteria used are:

Relative advantage: An analysis of the technical and business superiority of the innovation over technology it might replace.

Compatibility: An analysis of how well the innovation meshes with existing approaches and techniques that are routinely used.

Complexity: An analysis of how easy the innovation is to understand and use. Complexity is usually reflected in both the nature and extent of training and the manifestation of any new tool support features.

Trialability: An analysis of the type, scope and duration of feasibility experiments and pilot projects.

Observability: An analysis of how easily and widely the results and benefits of using the innovation are communicated.

Transferability: An analysis of the economic, psychological and sociological factors that either impede or aid adoption. Principal factors are prior technology drag, irreversible investments, sponsorship, and expectations.

4.1 Survey Cases

The survey [11] organized the twelve industrial case studies into three clusters (commercial, regulatory, and exploratory) and reported on a series of findings on each cluster. Additionally, subjective evaluations of the impact of formal methods were made for each case study in terms of vectors indicating a positive (+), neutral (0), or negative (-) impact of formal methods. The appended table summarizes the cases and evaluations.

The Commercial Cluster consisted of five cases in which the profit motive was paramount. Products developed by IBM, Praxis, InMos, Tektronix, and HP were evaluated. The Regulatory Cluster consisted of four cases in which safety- and security-critical regulatory agencies were involved and high assurance a necessity (TCAS, Multinet Gateway, Darlington nuclear reactor software, and Paris subway control system). The Exploratory Cluster consisted of three cases in which experimentation was the prime motive.

4.2 Evaluation

In this section, we present an introductory analysis of the survey data from the perspective of the adoption framework.

4.2.1 Relative Advantage

In general, the survey [11] shows that the use of formal methods had a generally positive impact on most key evaluation criteria relevant to relative advantage across the three clusters. The use of formal methods clearly enhanced the ability to demonstrate increased assurance in safety-critical systems over existing methods, thereby aiding certification by the regulatory bodies. In the case of the Commercial Cluster in terms of key commercially-oriented criteria (e.g., cost, quality, time-to-market) formal methods were marginally positive (with one exception of inappropriate application of an immature method). The observed marginally positive benefit may understate the relative advantage of formal methods because of the first-time nature of the application (learning curve). In terms of lifecycle stages, formal methods showed a net positive impact.

4.2.2 Compatibility

There is limited evidence in the twelve cases studied of formal methods successfully integrating with existing development methods or the existing installed base of tools and techniques. Several of the projects studied made the choice to work apart from the conventional software development process.

Only one of the cases (SSADM) attempted (and succeeded) to integrate formal methods with another method (Object Oriented Design). In the CICS case, the existing IBM software development process was modified by adding an additional step in the then current IBM process architecture. In the remainder of the cases, the formal methods activity was either carried out separate from or in parallel with the conventional development processes.

4.2.3 Complexity

The evidence in the survey indicates formal methods as currently crafted are difficult to understand, and are notationally complex and complicated. It also showed that moderate (on the order of a few weeks to a few months) levels of education and training, when coupled with follow-on work on real projects, considerably helped understanding of such formal artifacts as notation, specification language structure, and rudimentary proof checking. In contrast to many formal methods research projects the cases studied in the survey did not as a rule involve deep or scientifically hard problems.

4.2.4 Trialability

Formal Methods are relatively easy to experiment with, and there are no unduly expensive materials or tools to buy in order to get started. Several of the cases surveyed showed that experimentation and pilot projects played a major role in furthering the adoption of the formal method used in the serious development projects.

4.2.5 Observability

Formal Methods appear to be more observable in the negative than the positive. The degree to which knowledge of the results is transferred outside the group to a wider software engineering audience or general system engineering community has traditionally been only when the results were negative. Software engineering, as a discipline, does a relatively poor job of systematically surveying results of important projects, relying instead on grapevine reporting of anecdotes.

With respect to what groups outside of the specific project group learned or knew about the results, it appears there was limited awareness of the formal method activity in any substantive sense. This fact would seem to argue that results of formal methods use cannot be communicated well or easily.

4.2.6 Transferability

The survey data indicate that Formal Methods suffer from prior technology drag and the perception of irreversible investment, but have also benefited from sustained sponsorship. FM has largely lost the expectations game.

Prior Technology Drag: Somewhat paradoxically, in several cases (IBM CICS, Paris subway, COBOL Structuring Facility, InMos, HP) prior technology acted as a stimulant rather than a drag to the adoption of the formal method. The innovation represented by formal methods either represented a way to gain intellectual control over an inadequate legacy system in order to re-engineer it, or to master a new technology with which the organization had no prior experience.

Irreversible Investment: The perception on the part of potential new adopters of irreversible investment is definitely a major impediment to the adoption of formal methods. This situation is an example of where perception and reality diverge. The survey data show the up-front costs in tools and education for all twelve cases were relatively minor. roughly comparable to the costs for any new method or technique. While not trivial, the evidence shows they are not onerous. Tool support is a relatively minor cost, with basic tools to assist in writing and reading formal specifications available for a few hundred dollars to up to as much as $1000 for proof checking or theorem provers. It should be noted that these tools are far from industrially rugged when compared to CASE products.

Sponsorship: Government sponsorship has figured strongly in the development of formal methods. This sponsorship role has been two-fold: first, as a prospective (and eventually actual) user of the technology; and second, as a subsidizer of early adopters. The first role has largely been successful in bringing about necessary developments in mathematical theories, logics, automated tool support, and education and training. The effectiveness of the government in the second role is somewhat more problematic. On the one hand, the goal of providing the means of developing better software as a result of applying more scientific rigor to what is otherwise a largely ad hoc activity has had an impact in the fields of security and safety-critical systems, fields in which the responsibilities of governments as evaluators and certifiers is undisputed. On the other hand, this government perspective has for the most part ignored outside market and industrialization trends which have been evolving over the past decade in more sophisticated environments of tools, changing demands for accessibility by users, lower cost structures, different programming and development process paradigms. The effect has been to retard the ability of early adopters to integrate with the larger software engineering community.

Expectations: Evidence in the survey shows FM has not benefited from positive expectations that it would become a dominant and widely used technology. Further, FM has suffered from negative expectations as an a priori impediment [12]. Given that expectations involve psychological and sociological attitudes of the potential adopter community (in this case programmers and software engineers), there is a notable faddishness to the number and attributes of entrants into the better software business, which accompanied with marketing publicity, creates a "honeymoon" period during which the innovation takes off. Also, grapevine reviews, aided in recent years by the explosion of networked user groups, have spread a variety of mostly negative messages. For the most part, formal methods has not experienced a honeymoon period enjoyed by other software engineering methods. Almost from its inception as a discipline, formal methods has been viewed by the mainstream software development industry in largely negative terms. At best, FM is perceived as an evil that is either to be avoided (because of cost, education and overselling) or, if mandated (as per government requirements), then carried out in damage limitation mode.

4.3 Possible Solution Paths

Having identified adoption impediments, there is a responsibility to suggest some ideas on possible ways to overcome them. Certain solutions are suggested by the cases themselves. These recommendations are preliminary as we have yet to complete our analysis.

4.3.1 General Recommendations

Various general fixes are suggested:

- Development of improved tool interfaces. We need both more familiar interfaces (e.g., Windows) and interfaces that keep pace with evolving other computer technology trends such as visualization, animation and multimedia.
- Development of robust and reliable tools. These tools should incorporate ideas from CASE tools and SEEs, such as version control and configuration management.

- Development of better notations that are expressive and accessible to larger user base.

4.3.2 Adoption Criteria Recommendations

In addition, the adoption analysis also suggests possible routes to overcoming the specific problems highlighted in the adoption criteria.

Compatibility: There are several ideas concerning how formal methods might find pathways for compatibility. With respect to integrating with existing methods and processes:

- The existence of a rigorous process architecture provides natural insertion points for formal artifacts (e.g., CMM levels 3 and above, or, as in the SACEM case, as an additional step).
- The searching of compatibilities between existing techniques or methods and elements of formalization (e.g., the work of Jackson and Z ave linking JSD and FM), or common aspects of object orientation and formal specification (as in the SSADM case).
- Developing hooks into existing CASE tool suites.

Observability: One means of improving observability, is through such efforts as the survey. Other ways to increase observability are through inclusion of internal software education and training programs (such as the IBM SWE Workshops) and better exploitation of publicity mechanisms such as newsgroups, industry association meetings and the trade press. On the more technical side, the survey reported the lack of appropriate relevant metrics for FM projects. Development of such metrics, particularly if they were integratable into current measurement practices used in industry, should help facilitate making the results more accessible.

Transferability: Overcoming prior technology drag requires a deliberate effort to link formal methods with meaningful change in industry, either through re-engineering efforts or quality assurance. Often this involves mundane problems which are not viewed by many in the formal methods community as challenging. As a number of the cases surveyed show, these mundane problems have the advantage over challenging research topics of either representing a financially meaningful revenue stream or an installed base that does not make business sense to jeopardize. In addition, as a starting point, mundane problems may have a better chance of demonstrating successful results, which can lead to more challenging work.

References

[1] Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase and Mark Saaltink. "EVES: An Overview." In *Proceedings of VDM'91*, Noordwijkerhout, The Netherlands, 1991.

[2] Dan Craigen, Sentot Kromodimoeljo, Bill Pase, Mark Saaltink and Irwin Meisels. "The EVES System." In *McMaster International Lecture Series on Functional Programming, Concurrency, Simulation and Automated Reasoning*, Peter Lauer (ed.), Springer-Verlag, 1993.

[3] David Guaspari, Carla Marceau and Wolfgang Polak. "Formal Verification of Ada Programs." In IEEE Transactions on Software Engineering, pp 90-97, September, 1990.

[4] R.S. Boyer and J Strother Moore. "A Computational Logic." Academic Press, 1979.

[5] R.S. Boyer and J Strother Moore. "A Computational Logic Handbook." Academic Press, 1988.

[6] S. Owre, J. Rushby, N. Shankar and F. von Henke. "Formal Verification for Fault-Tolerant Architectures: Some Lessons Learned." In Proceedings of Formal Methods Europe'93 (FME'93), pp 482-500. Editors J. Woodcock and P. Larsen, Springer-Verlag 1993.

[7] S. Owre, J. Rushby and N. Shankar. "PVS: A Prototype Verification System." In Proceedings of the 11th Conference on Automated Deduction (CADE'11), pp 748-752. Editor Deepak Kapur, Springer-Verlag 1992.

[8] N. Plat. "Experiments with Formal Methods in Software Engineering." 1993.

[9] J. Spivey. "The Z Notation: A Reference Manual." Prentice Hall, 1992.

[10] M.J.C. Gordon, 'HOL: a Proof Generating System for Higher Order Logic', in: VLSI Specification, Verification and Synthesis, edited by G. Birtwistle and P.A. Subrahmanyam, (Kluwer Academic Publishers, 1988), pp. 73-128.

[11] Susan Gerhart, Dan Craigen and Ted Ralston. "Experiences with Formal Methods in Critical Systems." *IEEE Software*, January 1994.

[12] J. Anthony Hall. "Seven Myths of Formal Methods." *IEEE Software*, September 1991.

Appendix: Table of Evaluation Ratings from the International Survey of Industrial Applications of Formal Methods.

Table 1: CGR Evaluations of Regulatory and Commercial Cases

	Darlington	Multi Net	Sacem	TCAS	ssadm CASE	IBM/ CICS	cobol/ sf	Tek	Trans puter
Client Satisfaction	+	+	+	+	NA	NA	NA	+	+
Cost (of product)	NA	NA	NA	NA	NA	NA	NA	NA	+
Impact	+	+	+	NA	NA	+	NA	+	NA
Quality	0	+	+	NA	+	+	+	+	+
Time-to-Market	NA	NA	NA	NA	NA	NA	NA	+	NA
Cost (of process)	-	NA	0	NA	NA	0	+	0	+
Impact (of FM process)	0	0	+	+	+	+	+	0	+
Pedagogical	+	+	+	+	+	+	+	+	+
Tools	NA	0	+	NA	-	+	0	0	+
Design	+	+	+	+	+	+	+	+	+
Reuse	NA	+	+	NA	NA	NA	NA	+	+
Maintenance	NA	NA	NA	NA	NA	+	+	NA	NA
Requirements	+	+	+	+	NA	+	NA	+	0
V&V	+	+	+	NA	+	+	+	NA	+

Real-Time Software and Proof Architectures

August 18, 1994

Pam Binns
Honeywell Technology Center
Minneapolis, MN 55418
(612) 951-7414
binns_pam@htc.honeywell.com

Introduction

An important class of formal methods are those applied to the real-time behavior of computer systems. Since no single theory deals with all aspects of a system's behavior, an integration of existing theories must be achieved to simultaneously address different real-time requirements of complex systems. Different real-time scheduling theories provide implementation techniques and analysis algorithms that allow aspects of timing behavior to be formally analyzed, providing increased assurance of temporal correctness. An integration of hard real-time scheduling techniques with stochastic methods for soft real-time requirements and behaviors is needed, coupled with concurrent state machine models of the implementation.

Methods and associated tools that allow a systematic and automated application of these techniques are needed and are possible. In particular, our work focuses on developing integrated scheduling theories that provide temporal semantics for high-level specifications. These specifications can be fed to tools that automatically configure and populate a real-time executive, automatically generate real-time schedulability models of the configured system, and automatically solve the schedulability models. We outline what we call a proof architecture, which is a reusable (parameterized, configurable) proof outline that integrates various formal methods needed to verify the real-time properties of a particular application built according to a reusable software architecture.

The details of our current work in real-time software architectures provide an example of what is a more general approach. Software architectures are evolved (often from an existing reusable software architecture) along with an associated proof architecture. These architectures are reusable because they can be reconfigured for a large number of applications within a given domain, and possess a formal semantics provided by the various formal methods that have been selected and integrated. Analyses and arguments of correctness are then obtained largely by "plug-and-chug", where much of this can often be automated.

Real-time Application Requirements

Almost all complex embedded systems must provide traditional hard real-time feedback control and must also handle events that can best be characterized stochastically. Some control systems are safety-critical, and a sound formal basis for scheduling that supports analysis and verification is highly desirable and will increasingly be required.

Systems that control continuous physical processes depend heavily on the hard real-time scheduling of periodic tasks, where the exact requirements for a particular periodic task set must be arrived at in conjunction with the control engineer. In general, the tasks may all have different periods, specific deadlines, or may produce outputs or consume inputs at points other than the beginning or ending of their executions. Tasks may also need to synchronize access to shared resources or need to be distributed among multiple processors to meet the computation requirements. It must be possible to handle transient processor overloads due to variations in task execution times, etc.

Complex systems must also handle events as well as manage continuous processes. Physical processes correspond to a variety of different events (e.g. valves open and close, humans interact with the system, targets and threats appear and disappear, system components fail). This translates into a requirement that a system be able to support aperiodic tasks, which is to say support computations that occur in response to specific events. In some cases an event must be responded to within a fixed hard deadline, e.g. target proximity detection for missile fusing, ground collision warning for terrain following flight direction. In some cases the response to an event can best be characterized stochastically for a class of events, e.g. 98% of the time a dialtone will be obtained within 4 seconds of lifting a telephone handset.

What is typical in practice is that many real-time requirements are not directly specified. Instead, they are derived during development from specifications of system functionality and quality of service. Real-time requirements are often difficult to identify, sometimes don't appear as explicit system requirements, and are extremely difficult to verify.

Real-time behavior is a property of the overall system, not a property of individual components, and must be dealt with at the systems level. There are often interdependencies between different real-time requirements that result in poorly understood design trade-offs and interactions between components. Such interdependencies are sometimes manifest as complex trade-offs between functionality at the system specification level, and can lead to extremely subtle defects having wide-ranging and possibly catastrophic effects.

Real-Time Software and Proof Architectures

The real-time requirements of large complex systems are diverse and often competing, and no single analysis or proof technique or tool is appropriate for the verification of all of a system's real-time requirements. What we will do is list some formal methods of use in our current work on real-time software architectures, sketch out a configurable software architecture based on these formal methods, then introduce the idea of a proof architecture by example. As the software architecture is evolved (populated, specialized) to create a specific application, the proof architecture is simultaneously evolved to provide analysis and increased assurance of correctness of the timing properties of that application. The systematic way in which this is done is automatable, at least to a significant extent, as we have already demonstrated for some of this process in our ARPA/ONR Domain-Specific Software Architectures program [1].

Rate monotonic analysis allows one to analyze the behavior of a set of fixed priority preemptable real-time processes sharing a processor. The basic theory deals with periodic processes each having specified bounded compute times, intervals between dispatches, deadlines relative to each dispatch, and interactions with other processes. This is the model provided to users of our automatically configurable executive, who specify the collections of real-time processes (and associated source modules) they wish scheduled, together with the patterns of communication and resource sharing among those processes.

Concurrent state machine models and related methods of process algebras and temporal logics provide powerful notations for specifying complex discrete state event-driven systems, together with methods for demonstrating liveness and safety properties for a specified system. The design of our configurable executive is based on such a model. State machine models are used at a process level, a mode level, and a processor level. At the process level, the state machine is described by a "life cycle" with transitions among process states of stopped, initializing, suspended, and executing. Different modes of operation are supported by allowing varying subsets of processes to active at different times. At the mode level, run-time mode transitions are modeled as state transition models that must interact with the various process state machines. At the processor level, different instances of a configured executive execute on different processors, where these concurrent state machines go through various power-up, operation, consensus protocol, and shut-down states.

Our process level concurrent state machine implementation model executive is different from that of CSP (Ada, Occam) in that state machines are not processes in the traditional sense. Instead, events correspond to things like clock interrupts, service calls made by processes, etc. States are predicates over internal kernel variables (e.g. which queue contains a process, accumulated process execution time). Transitions are event-triggered invocations of nonpreemptable subprograms that manipulate the internal kernel variables. Traditional Floyd/Hoare sequential proofs can be used to show that a piece of source code correctly maps a program state in which a given precondition is true to a program state in which a given postcondition is true. Process level transitions are the result of event-invoked subprograms with preconditions defined by the predicates of the state prior to the event and post conditions by the predicates for the the state resulting from the event.

A user wishing to create a specific real-time application defines a set of processes and their associated timing requirements along with a set of modes of operation and the transitions among them. This is done using an architectural description language called MetaH. The information in the MetaH specification is then used to specialize source code templates for the configurable executive and a real-time schedulability model. We have not formally captured the concurrent state machine design or state predicates using any toolset yet, but the idea would be to similarly specialize model templates for such tools using information in the specification.

The source code and model templates are configured by inserting the proper number of processes of the proper type (where a process type in MetaH corresponds to the state machine that describes the scheduler state transitions that a process can undergo at run-time). The mode transition diagram must also be inserted. Values like periods, compute times and deadlines are substituted for the appropriate parameters in the various templates. Evolutionary model specification and associated software development is supported by allowing partial specifications to be generated and analyzed for temporal correctness. The analysis techniques in our current proof architecture system have been applied to specifications with several dozens of processes and are designed for applications with a few hundred processes on a multiprocessor system.

Proof Techniques for Hybrid Scheduling

Among all real-time requirements of a complex system, accurate and predictable scheduling is among the most important. Systems that are not designed and analyzed using sound methods are subject to unpredicted and unmanaged transient overloads and/or thrashing. These situations sometimes have an unfortunate tendency to occur at the worst possible moment (e.g. fault event handling, appearance of multiple threats, "alarm storms"). In this section, we look at how the principles of a proof architecture apply to a more detailed and less well understood problem called hybrid scheduling. Hybrid schedules include mixtures of periodic, hard deadline aperiodic, and soft deadline aperiodic traffic.

We are working towards the development of integrated analysis and implementation for hybrid scheduling in MetaH, which currently supports periodic processes using fixed priority preemptive scheduling and aperiodic processes using the deferred server method. We are working to extend this with slack-based implementation techniques for aperiodics. Slack-based analysis will be used for hard deadline aperiodics [2,4], and we plan to integrate more traditional aperiodic analyses (e.g. queuing theory, Petri Nets [3], or more generalized stochastic processes) for large scale approximations of non-critical traffic and time averaged system behavior for underlying slack servers.

From the user's perspective, a set of aperiodic and periodic processes are specified. The periodic task set is first analyzed alone for schedule feasibility using our linear schedulability analyzer [5,6]. Then the maximum slack, or equivalently the maximum amount of utilization that can be consumed while guaranteeing the deadlines of periodic processes, is computed. The aperiodics are also priority ranked relative to one another and to the periodics. Worst case compute times, any deadlines, and arrival assumptions must also specified for aperiodics. (The precision in the specification of arrival assumptions will depend both on the traffic stream's requirements and on the criticalness of the requirements.)

Slack values are then computed at each priority level, and slack availability for the specified requirements is assessed across the priority levels. It is sometimes permissible to allow a periodic to miss a deadline occasionally to favor a high priority aperiodic. Our current tool provides analysis values that allow an assessment of possible impact on a periodic by an aperiodic by examining the compute time variations that can be tolerated at run-time. Our current and planned algorithm and analysis techniques provide some flexibility and support for trade-offs between the various performance characteristics of an application (e.g. periodic deadlines versus aperiodic response times, average throughput versus the risk of missing a particular deadline), and we have identified additional forms of such sensitivity analysis information that may be useful.

Using slack-based methods, it is possible to automatically handle the low-level details of scheduling. This frees the system developer from the need to construct timelines for all possible system scheduling behaviors. The models and executive implementations are developed together, and execution times used in detailed analysis for aperiodics will also include implementation overheads such as context swaps, scheduler overhead times and inter-processor synchronization and communication. In a slack-based implementation, a configured executive contains a table of statically computed slack values. Slack values are determined from the periodic workload, and in some cases higher priority aperiodic workloads. Slack counters are maintained and updated when the kernel changes state as described in the previous section. Unused execution time (compared to the worst case specified) is easily reclaimed and made available as slack bandwidth to aperiodics.

Our approach emphasizes maintaining detailed consistency between design, analysis and implementation, so that the analytic results provide assurance of implementation correctness throughout the software evolution process. Our detailed design-time analysis of mixed periodic and aperiodic (hybrid) workloads can be extended and used in conjunction with other analysis tools, both to estimate the likelihood of certain kinds of anomalous behaviors and to study the feasibility of various design alternatives. For example, further work could provide analysis capabilities that allow us to more accurately examine the tails of distributions. This is needed for analyzing system behavior that is not near the mean, e.g. probabilities of missing deadlines, probabilities of queue overflow, probabilities of data loss. These analyses are especially difficult and especially necessary when modeling the reliability of complex systems, and in making design evolution trade-offs that involve system reliability.

Our reusable real-time architectures then consists of the collection of source code and formal model templates. Acquiring the associated tools, developing the templates, and developing the methods for configuring the templates in a way that assures their mutual consistency are complex tasks subject to some of the usual criticisms of formal methods: they require high levels of specialized skills and they take a large amount of effort. However, once these assets are obtained, they can be easily reused on many, many applications. Moreover, the specialization of the templates and invocations of the analysis tools can be largely automated. By developing a highly reusable software and proof architecture, the original expense of applying formal methods can be amortized across a large number of applications.

References

[1] Pam Binns, Matt Engleharl, Mike Jackson and Steve Vestal, "Domain-Specific Software Architectures for Guidance, Navigation and Control", Honeywell Technology Center, Minneapolis MN, 1994.

[2] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems", *Proceedings of the Real-Time Systems Symposium*, December 1992.

[3] V. Mainkar and K. S. Trivedi, "Transient Analysis of Real-Time Systems Using Deterministic and Stochastic Petri Nets", Duke University, Dept. of EE, Technical Report, 1994.

[4] S. Ramos-Thuel and J. P. Lehocky, "On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems", *Proceedings of the Real-Time Systems Symposium*, December 1993.

[5] Steve Vestal and Pam Binns, "Scheduling and Communication in MetaH", *Real-Time Systems Symposium*, December 1993.

[6] Steve Vestal, "Fixed Priority Sensitivity Analysis for Linear Compute Time Models", *IEEE Transactions on Software Engineering*, April 1994.

A FORMAL MODEL OF PROBLEM SOLVING AND ITS IMPACT ON SOFTWARE DEVELOPMENT *

by
Daniel Cooke
University of Texas at El Paso
El Paso, TX 79968-0518

1.0 INTRODUCTION.

There appear to be at least two problems concerning formal models for software development. These problems contribute to the separation of the associate theory and practice. The first problem is that many formal models impact only a small portion of the software life cycle. Therefore, the cost/benefit ratio for the **micro-models** is unfavorable (i.e., the **utility** of a model is often low).

To address this issue, many broader models[1] actually incorporate several models, where each constituent model addresses a portion of the life cycle. The solution to the first problem leads to the second problem: the fact that **macro-models** often incorporate several models, requires the user to understand several different abstractions and the interactions among them. The resulting complexity may be perceived as a situation where the cure is worse than the disease. A worthy goal appears to be a formal model that addresses a larger share of the software development cycle in a single abstraction.

In order to be more useful the model must provide support beyond the software development phases and address operational issues such as reliability. This paper presents three considerations which affect the utility of a formal language/model[2] for software development. First, languages must be more readable and writable so that developers can be more effective when producing software. Second, languages should, in a single, uniform abstraction, allow the developer to express as many aspects of a problem solution as possible, including constraints on the software's behavior. Finally, languages should support software evolution. The potential impact of a formal model for problem solving based on the language, BagL, is discussed herein. By showing how BagL addresses the three considerations above, one may observe how the BagL model may impact a larger share of the software life cycle and how BagL may lead to improvement in (1). the effectiveness of software developers and (2). the reliability of their products.

2.0 ABSTRACTION AND REPRESENTATION.

If one can make a software developer more effective, then it is possible to favorably impact the reliability of complex software products. Software developers are more effective when using more readable and writable languages to express software solutions.

In order to improve upon the readability and writability of software solutions, suitable abstractions and associated representations are necessary. One approach is to abstract out of current programming languages major sources of complexity and do so without ending up with an inherently ambiguous language.

---

* Research sponsored by the Air Force Office of Scientific Research (AFSC), under contract F49620-93-1-0152; NASA, under contract NAG 2-670 Supp. # 2, and by NSF grant No. CDA-9015006.

[1] As an aside, one must realize that most current models assume a software life cycle more or less like the traditional waterfall model. However, suppose there exist different formal approaches to problem solving which would actually replace the traditional life cycle with a new one. Such a formal approach might result in an abbreviated life cycle (compared to those we now use) and impact a much larger share of the new life cycle.

What could cause such a radical alteration in the software life cycle? In this paper, it is hypoth esized that a true shift in language paradigm would result in a new software life cycle. The current life cycle has evolved around the imperative programming language paradigm. A new paradigm might likely re quire a new life cycle for software development. For example, the rapid prototyping model suggests a life cycle much different from the current life cycles. [Luqi]

[2] Here it is assumed that a formal language implies a model for problem solving.

2.1 ABSTRACTION.

"Since Pratt's paper on the design of loop control structures was published [Pratt] more than a decade ago, there has been continued interest in the need to provide better language features for iteration."[Bishop] The need for better language features is due to the fact that the construction of looping algorithms is very complex and costly.[Mills]

In order to establish better abstractions for loops, it is necessary to identify the fundamental purposes served by loops. Iterative control structures are employed in order to process nonscalar data structures.[Bishop] The three basic purposes of loops are:

1. To produce nonscalars from scalars (e.g., Fibonacci sequence);
2. To produce a scalar from a nonscalar (e.g., summation);
3. To produce a nonscalar from a nonscalar (e.g., squaring a matrix).

[Cooke91]

In the same way languages like Algol and Pascal introduced fundamental control structures in order to limit the degrees of freedom in solving problems with GOTO-based languages, the research leading to BagL has attempted to introduce fundamental constructs for processing nonscalar data structures. This research has uncovered that there are five ways in which nonscalar structures are processed: the **regular**, **irregular**, **generative**, **eventive**, and **transitive** processing of nonscalar structures. BagL possesses constructs for each of the five types of nonscalar processing. Thus, the BagL abstraction effectively replaces iteration and recursion with a small number of constructs for processing nonscalar data structures.

BagL is a formal, executable specification language which provides for database and scientific computing in a uniform level of representation. Using BagL, a problem solver provides little in the way of algorithmic detail in a problem solution. Instead, the problem solver describes the solution directly by specifying, via a metastructure, the data structures which will hold results useful in solving a problem.

A metastructure is a single, general purpose structure which is configurable into any possible data structure. The structures of BagL are persistent, in that there is no need for the programmer to write routines which convert an external database (i.e., one stored on disk) into an internal database (i.e., one existing in memory) or vice versa [Lamb].

BagL is a small language, in that there is a small number of language constructs. All BagL constructs interact well; all constructs are defined to operate only on bags, and all constructs produce, as results, only bags. The denotational semantics of BagL were completed in the spring of 1993. A prototype interpreter based upon the semantics was written in the summer of 1993 and is currently being revised. BagL possesses the simplest possible computational model. There is nothing in BagL's computational model that does not absolutely have to be there. By abstracting out a major source of complexity, BagL offers an abstraction which should improve the effectiveness of problem solvers.

2.2 REPRESENTATION.

The representation of a problem solution in any language, regardless of the abstraction level of the language, is complicated by two needs:

(1). the need to present nested operations and
(2). the need to select operands upon which the operations are to apply.

Nested program structures are a major source of confusion when attempting to read or write a program. In procedural programs this source of complexity is made worse by the fact that there are many different types of control structures that can be nested, including procedures, functions, iterative statements, and if-then-else structures. Programmers must understand the interaction between nested statements of differing constructs (e.g., loops inside if-then-else, etc.) and then they must understand the interaction of the non-control statements inside the nested structures.

BagL presents an improvement over the procedural languages in that **only** non-control statements are nested in BagL (Control statements, other than guarded commands, do not exist in BagL). Therefore, the programmer need only understand the interaction of nested terms and relations in BagL

-- nothing else can be nested. However, BagL still suffers from some of the confusion which results from nested structures. Consider the irregular BagL function to compute the square of a matrix:

function square(dom(t), ran(sq)) **is** [+([*([t(i, (*)),t((*),j)])])]

This function is indeed daunting. It states that for each **i** and **j** associated with table **t** multiply row **i** by the corresponding elements of column **j**. This operation results in a bag of products to which the + sign is distributed, resulting in the singleton bag corresponding to the **i,j**th position of the range bag, **sq**.

With the **square** function, one observes the difficulty of representing nested structures; we call this problem the **containment** problem. What is inside what? The typical approach to dealing with this problem is to use a directed graph to represent the flow of the equation. [Berztiss] See figure 1.

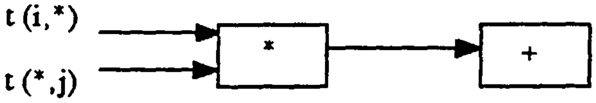


Figure 1. Nested Structures.

While an improvement over the textual definition, there may be better ways to indicate the nesting of BagL terms. Using our method, the containment of an operation is self-evident. Consider again the BagL function to square a matrix, but this time the presentation will benefit from an improved visual approach:

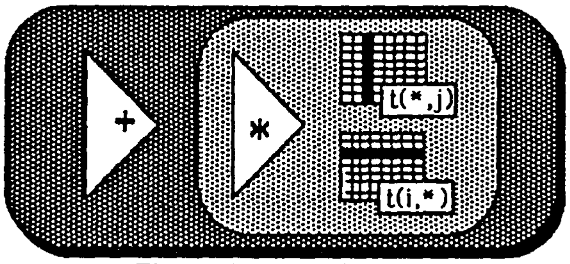


Figure 2. Containment.

As one can observe, the containment of an operator is self-evident when compared to the textual function, above. BagL's abstraction actually facilitates the development of a visual interface. The research into visual languages and interfaces is typically based upon current paradigms of programming. The languages serving as a foundation for this work have complicated computational models which tend to muddy the waters in the search for interface improvement. As stated previously, BagL has the simplest possible computational model. There are no unnecessary complications in the model.

Similarly, the BagL model suggests basic operations associated with the selection of operands for computation. There seem to be basic selection operations which may also have revealing visual definitions. The two fundamental ways to select data are (1). based upon value and (2). based upon position. The darkened row and column of tables **t** in figure 2 suggest a scheme to depict the selection of operands based upon position. With a better representation scheme, BagL should provide a view of problem solution that is easy to read, write, and comprehend. With better comprehension should come increased effectiveness.

3.0 INTEGRITY CONSTRAINTS.

BagL can be extended to impose database integrity constraints. The extension we are developing will not add to the knowledge required of the BagL programmer. Functions which impose integrity constraints behave in a manner consistent with the behavior of BagL functions which compute results. Consider a simple example. Assume a database exists, like that presented in figure 3, view 1.

A BagL function to compute a ten percent raise for all employees, as applied to the Employee Database, would be:

Function Raise(Dom(Salary), Ran(Salary)) **is** [Salary * 1.1]

One type of BagL function, called an **outer function**, is initiated for execution based upon the availability of data in the database. In particular, the domain variables of the function (e.g., Dom(Salary)) must be paired with values in the database to which the function is applied. (The data dependent execution strategy provides the BagL eventive construct.) When the function begins execution, the domain variables are consumed from the database. Assuming **Raise** is an outer function, the initiation of its execution results in the change to the database as it is depicted in view 2 of figure 3.

View 1

	Name	Salary
Employee Database	bob	50,000
	mary	55,000
	sue	90,000

View 2

Name
bob
mary
sue

View 3

Name	Salary
bob	55,000
mary	60,500
sue	99,000

Figure 3. Modifying the Employee Database.

It is possible for domain variables to be protected from consumption. It is also possible to initiate functions in the more traditional way, via function references (or invocations). Given that function **F** is invoked by some other function **G**, the function **F** is called an **inner function**. Function **F** receives its arguments from and returns its result to the invoking function **G**. When an outer function completes execution, its results are paired with the range variable(s) and placed in the database. For example, when function **Raise** completes execution, its result is produced in the database and paired with the name **Salary**. See figure 3, view 3.

The **Raise** function contains the term **Salary * 1.1**. In BagL, when a scalar (like **1.1**) is paired with a nonscalar (like **Salary**) the scalar is "normalized" to the length of the nonscalar and the associated operator is distributed among the corresponding elements of the associated structures:

```
[ 50000, 55000, ... , 90000 ] * [1.1]                     becomes
[ (50000 * 1.1), (55000 * 1.1), ... , (90000 * 1.1) ]     which becomes
[ 55000, 60500, ... , 99000 ]
```

In order to impose constraints on a database, it is typically incumbent upon the programmer to distribute appropriate guards into all functions which update a constrained field. For example, suppose there is a cap placed on salaries in the Employee Database above, limiting salaries to be no greater than $100,000. To impose this constraint, the programmer in a typical language must place an appropriate guard on any statement updating the salary attribute. Furthermore, the programmer must be concerned about **indirect** updates to the constrained attribute. For example, when a new employee's information is added as a tuple to the Employee Database, the programmer must recall that there is a constraint on **Salary** and furthermore, the programmer must recall that **Salary** is an attribute of the Employee Database.

The problem with relying on the programmer to impose constraints is that the programmer has to keep track of the constraints and know when it is appropriate to impose them. When a constraint changes, a system maintainer must remember all of the functions which are affected by the change and make the appropriate modifications.

One can envision BagL functions which initiate execution upon availability of databases rather than upon domain variables. We are making simple alterations to the syntax and semantics of BagL so that constraint functions will behave and be written in a manner consistent to BagL computational functions:

```
Constraint(Dom(DB,DB'),Ran(DB)) is
        [       [DB']  when <=(Salary',100,000)
                [DB]   otherwise          ]
```

This function executes whenever the named Database and its successor (denoted by a prime) is available. Whenever the named database is updated (i.e., when both DB and DB' become available), the function, **Constraint**, returns the new database if the Salary constraint is met. Otherwise, the old database is returned. There is no need for the programmer to place guards throughout a program or set of programs which affect, either directly or indirectly, the constrained attribute. The constraint is stated one time and is imposed on **any** update, whether direct or indirect. If, at a later time, the constraint changes, only one software update is necessary. By removing the responsibility for imposing constraints from the programmer, BagL provides a foundation for the production of more reliable software.

4.0 SUPPORT FOR SOFTWARE EVOLUTION.

The reliability of a program is subject to **environmental** influences. Programs do not operate in a vacuum. They are expected to fit into some environment and operate reliably in that environment. Unfortunately, environments change and specifications which were true in the original environment may not be true in future environments. Environmental changes are indicated when a system is presented information which is inconsistent with the program's knowledge. A program cannot respond correctly when environmental information contradicts specified information.

In [Luqi and Cooke] it is shown that many environmental changes result in an inconsistency in the software system's specification. Using the integrity constraints discussed in the previous section, shifts in knowledge resulting in inconsistencies, are easily detected in constraint functions which operate on a DB and its successor, DB' (e.g., when some information is true in DB and false in DB'). Detecting inconsistencies permits the system itself to alert programmers of the need for adaptive maintenance. Thus, through its features for integrity constraints, it is possible that BagL may provide additional support for maintenance and, as a result, further improve upon the reliability of systems developed in BagL.

5.0 CONCLUSION.

Formal languages may lead to improvement in (1). the effectiveness of software developers and (2). the reliability of their products. To do so, however, languages must impact a larger share of the software life cycle. This paper presented three considerations which affect the reliability of software products. First, languages must be more readable and writable so that developers can be more effective when producing software. Second, languages should, in a uniform abstraction, allow the developer to express as many aspects of a problem solution as possible, including constraints on the software's behavior. Third, languages should support software evolution.

REFERENCES

[Berztiss] Alfs Berztiss, "The Query Language Vizla," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 5, (October, 1993), pp. 813-825.

[Bishop] Judy Bishop, "The Effect of Data Abstraction on Loop Programming Techniques," *IEEE Trans. Soft. Eng.* Vol. SE-16 Number 4, April, 1990, pp. 389-402.

[Cooke91] D.E. Cooke and A. Gates, "On the Development of a Method to Synthesize Programs from Requirement Specifications," *International Journal on Software Engineering and Knowledge Engineering*, Vol 1 No 1, (March, 1991) pp. 21-38.

[Luqi] Luqi, "The Role of Prototyping Languages in CASE," *Intl. Jour. of Softw. Eng. and Knwl. Eng.*, Vol. 1 Number 2, (June, 1992) pp.131-150.

[Luqi and Cooke] Luqi and Daniel E. Cooke, "Rapid Prototyping and a Model for Software Maintenance," in revision for *IEEE Transactions on Knowledge and Data Engineering*.

[Lamb] Charles Lamb et.al., "The ObjectStore Database System," *CACM*, Vol. 34, No. 10, October, 1991, pp. 50-63.

[Mills] 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.

[Pratt] T.W. Pratt, "Control Computations and the Design of Loop Control Structures," *IEEE Transactions on Software Engineering*, Vol. 4, No. 2, pp. 81-89, Feb. 1978.

Feature-Oriented Software Engineering

Herwig Egghart   Edgar Knapp

Purdue University
Department of Computer Sciences
West Lafayette, IN 47907-1398
{egghart,knapp}@cs.purdue.edu

August 27, 1994

Abstract

The long-term goal of our research is to reconcile the conflicting goals of increased software quality and shorter time-to-market. We have proposed a theory and a methodology centered around so-called features and have applied them to a software development project, the design of an SQL database management program. A corner stone of our approach is that it allows the software engineer to keep track of features throughout the life-cycle. This includes the reliance on incremental enhancements, the explicit specification of relationships between features, and the mapping of software features to the code that implements them.

We are also currently working on automatic support for our methodology. This includes the development of an interactive tool with hierarchical modeling capabilities, a browser for feature dependencies and interactions, an automatic version generator, and a consistency checker.

Keywords: Computer Aided Software Engineering, Configuration Management, Design Methodologies, Features, Feature Dependency, Feature Interaction, Life Cycle, Software Quality, Software Understandability, System Analysis and Design, Version Control.

1 Introduction

A pressing concern of the software industry is decreasing time to market without sacrificing software quality. Yet, the inherent complexity of modern software systems makes the goals of rapid delivery of applications and the attainment of high quality standards appear mutually exclusive. Much of the complexity of current software is caused by the abundance of features they support.

Many UNIX tools have upwards of 30 options, off-the-shelf commercial software has manuals totaling thousands of pages. To combat the inherent complexity of large software systems, it is necessary to break their structure down into manageable pieces. We see three dimensions in which this can be done:

Modular Programming, the oldest approach to hierarchical decomposition, attempts to organize system components around collections of related tasks.

Object-Oriented Programming, a more recent paradigm, organizes the data structures of a software system around collections of objects.

Feature-Oriented Programming, our contribution, aims at breaking down system functionality into collections of features.

While current modularization techniques seem sufficient to create a manageable first version of a software system, the complexity introduced by subsequent modifications and enhancements tends to get out of control. This problem has gained much importance through the increasing popularity of evolutionary life-cycle models [4, 6].

1.1 Vertical versus Horizontal Abstraction

The best weapon to combat the inherent complexity of large software systems is abstraction. Traditionally, higher-level notations have been used to abstract from implementation details. However, there are two problems with this vertical approach: First, it is difficult to verify the consistency between different levels of abstraction. Second, even if we abstract from all implementation issues, the external behavior of systems is becoming increasingly more complex so that it can no longer be easily understood.

The feature-oriented approach presents a horizontal alternative to the vertical abstraction paradigm. Rather than omitting implementation details, we investigate the omission of behavioral details, i.e. well-defined pieces of system functionality. Subsequent decomposition of these pieces leads to atomic features.

Horizontal abstraction induces an algebraic structure which gives a precise mathematical meaning to features. As a result, we can explicitly associate atomic features with software fragments ("marked fragments") and thus disable and enable features at will. This makes them traceable and manageable throughout the software life-cycle:

- During requirements analysis, our approach allows structuring and decomposing functionalities into features. All known feature interactions are also made explicit at that time.
- During design, internal system functionality is identified and managed in exactly the same way that external functionality was before.
- By mapping the functional structure into the actual source code, programmers are confined to implementing features in a clean and traceable way, which leads to understandable and maintainable code.
- For testing purposes, parts of the system functionality can be disabled, which allows testers to focus either on specific features in isolation or on feature interactions.
- Automatic evaluation of regression tests is usually difficult because the behavior of the system is not exactly the same as previously. By disabling functionality added in the meantime, the old behavior can be obtained at any time, although the internal structure of the system may have changed significantly.
- Marketing considerations sometimes require feature bundling in a way that was not anticipated during design. Instead of increasing the system complexity further to achieve the desired bundling, unforeseen functional subsets can be created easily with our approach.
- In order to train people in the use of a new product, it is often useful to start with a simpler, more understandable version of the system in which certain advanced features are disabled.
- Maintenance does not always mean *adding* functionality. Sometimes features have to be retired because they have turned out to be useless, poorly implemented, or even harmful. Our approach guarantees full reversibility, i.e. easy removal of obsolete or unwanted features.
- By using run-time feature control, a new approach to fault-tolerance becomes possible. If an error occurs within code belonging to a certain feature, only this feature needs to be turned off. The overall system may be able to continue operation at reduced functionality.

In [2] we illustrate most of these stages through the example of an SQL database management system. We give an analysis of its set of features, and show how this analysis has guided the design and implementation of the system. In addition, we show how readily available tools such as cpp (the C preprocessor), sed, and grep can be used to support selective enabling and disabling of features.

1.2 Related Work

We know of two previous attempts for obtaining better feature traceability:

Norman Wilde (University of West Florida and, like us, a member of the Software Engineering Research Center) has created the notion of "software reconnaissance" [8]. He runs test cases over an instrumented program to find out where in the program a certain feature is implemented. His reconnaissance tool tries to identify "unique code" with respect to the feature in question, i.e., program statements that are only executed when the feature is invoked.

Hart and Shilling (Georgia Tech) try to solve the problem earlier, viz. at the time a feature is being implemented [3]. They designed a syntax-directed editor that allows programmers to declare features and to link them explicitly with nodes in the program tree. Advanced operations like feature extraction (selected disabling of features) and feature instantiation (creating additional instances of generic features) are also possible.

What has not been attempted, though, is to define the *concept* of a feature. According to [3], *"it is precisely this lack of rigor which gives features much of their power. The designer is free to define as a feature any useful, easily described, slice of program functionality."*

While we agree that features may resist a *complete* formalization, we have found that a high degree of precision is desirable and achievable. Our feature-oriented methodology, which spans the entire software life-cycle, would not be possible otherwise.

The formal basis of our feature theory is elaborated in [1]. There we define features as sets of functional atoms that enjoy certain closure properties. Features give rise to a lattice of system versions with the bottom version possessing no features, the top version supporting all features, and intermediate versions implementing some consistent subset of features. We also illustrate the theory by applying it to a real-world program.

1.3 Organization of this Paper

After a brief introduction to feature-oriented terminology and feature theory (sections 2 and 3), we present our view of the evolutionary life-cycle in sections 4 through 9. We conclude with some thoughts on tool support and the role of formal specifications in feature-oriented software engineering.

2 Feature-Oriented Terminology

We define some feature-oriented terminology that forms the basis of the feature-oriented approach.

2.1 Features

Features of a system are found by imagining simplified system versions, which we call *reductions*. One can think of a reduction as a historic predecessor (which it might in fact be), or just as a preliminary version to be shown to the customer. The difference between a reduction and the full version is a *feature*.

Similarly, to determine whether some capability of our system is a feature, we try to imagine a reduction that would result from its removal. If such a reduction is possible and easier to understand than the original system, we have identified a feature.

The converse is not always true, though. One feature might logically depend on another feature, so the latter cannot be removed alone. But if the reduction becomes possible after removing some other features, we also know that we have a feature. (The full version inherits the features of all its reductions.)

2.2 Functionality

The overall system functionality is composed of three parts. They are core functionality, library functionality, and feature functionality.

Main input, main processing, and main output together form the core functionality, which would typically be the kind of view given in the first chapter of a user's guide. The core functionality must be preserved by every reduction, which means (by definition) that it is free of any features.

Library functionality is functionality that is not application-specific and is provided by the programming language, built-in procedures, run-time libraries, and other general-purpose tools.

Core and library functionality are completely free of features. All features are captured by the remaining feature functionality. In its entirety, feature functionality can be seen as one big composite feature, although in practice it is more useful to decompose it into individual features of a finer granularity.

3 Feature Theory

Not only is the feature-oriented approach of practical relevance, but it is also firmly grounded in theory. The following is a summary of some results of our feature theory.

Features are collections of atoms which in turn are atomic units of system functionality. Features satisfy a certain closure property, hence not every collection of atoms is a feature. The fact that singleton collections of atoms are features allows us to identify any such feature with the atom it comprises.

Two features can be classified as either orthogonal to ($\perp$), dependent on ($\succ$), or parallel to ($\|$) each other. Orthogonal features can be disabled and enabled independently. If orthogonal features are present simultaneously, feature interaction may occur. Feature dependency is an irreflexive, anti-symmetric, and transitive relation. Dependencies restrict the way in which features may be added or deleted. Parallel features cannot be disabled one after the other. They occur when features have some atom in common or when conflicting dependencies exist.

Versions are defined as downward-closed collections of atoms. Versions form a complete lattice with the full version as the top and an extremely reduced core version as the bottom. Any feature can be constructed as the difference between two versions, and any difference between versions is a feature.

Our version lattice is analogous to the version lattice of [5]. But they use it to model vertical abstraction, while we are interested in horizontal abstraction. Consequently, their partial order means "refinement" while ours means "simplification".

Features may be composed. It turns out that features form a monoid[1] under feature composition, which is defined as a restricted form of the union of features.

4 Feature-Oriented System Evolution

The feature-oriented life-cycle is evolutionary in nature. At the beginning, a core version is developed, which is subsequently incremented with feature functionality. Every increment is an atomic feature.

The feature-oriented life-cycle allows for maximum customer interaction. Every new feature the customer requests is inserted into a "wish list", which is transparent to the customer. At any time, the customer may put a new feature on the wish list, reorder priorities, or cancel a feature. Of course, it is in the interest of both the customer and the developer to keep the wish list as stable as possible. If the customer suddenly requests a feature that cannot be implemented with the present design, then the price for that feature will have to cover all the redesign and recoding effort.

Features from the wish list are processed according to their priority. First, the requirements engineer clarifies the semantics with the customer. If the feature is still on top of the wish list after that, the effort will be estimated and a price calculated. If the customer accepts the price and the feature is still on top of the list, the designer will devise an implementation strategy. Subsequently, it will be implemented and removed from the wish list. All these activities happen asynchronously, with the wish list acting as a pipeline.

Should the customer change his mind later, he can still cancel a feature after it has been implemented. In a feature-oriented implementation, features can be disabled and enabled at will. So if the customer changes his mind again, he can have the feature back.

Another advantage of automatic disabling is that a new version can be given to the customer every time a new feature has been implemented. Traditionally, an expensive synchronization is needed in order to obtain a stable and consistent version. All implementation and testing activity in progress must be finished, and nothing new may be started until the new version has been frozen. In a feature-oriented implementation, by contrast, features that are not ready for release can be disabled, and the customer gets one new feature and nothing else.

[1] A monoid is an algebraic structure with an associative operator and a unit element.

5 Feature-Oriented Specification

5.1 Core Specification

The first step to a feature-oriented specification is to specify the core functionality, which is equivalent to specifying a *core version*. The core version should be a useful approximation of the full version. It should capture the basic idea behind the system, but leave out any additional bells and whistles.

Everything else is based on the core specification, so it should be made as small, clean, and precise as possible. Since the core version is not "messed up" with features, it is far more susceptible to a formal specification than the full version. But even a semi-formal core specification will profit greatly from the simplification achieved by feature omission.

5.2 Feature Statements

The next step in the requirements specification phase is to decompose the feature functionality, i.e. the difference between the core and the full version, into features. This happens in two directions: Starting from the full version, we can remove features to get a sequence of reductions. Starting from the core version, we can add features to get a sequence of enhancements. It is a good idea to mentally construct each of the versions and to verify that they are logically possible. Every feature must also be checked to verify that its removal has a simplifying rather than an obscuring impact. When the two sequences meet at some version, we have completely decomposed the feature functionality into a set of mutually non-parallel features.

Whenever a feature has been identified (i.e. either something that has to be added to the core version to increase its complexity toward the full version, or something that can be removed from the full version to simplify it toward the core version), the question of atomicity arises. If the feature is quite complex but can be decomposed into simpler features by constructing intermediate reductions, then it should be further decomposed. Otherwise, decomposition halts and the feature is declared atomic.

Once the atomic features have been identified and named, their precise meaning must be defined. For each feature f the following question must be answered: "How does adding (or removing) f change the behavior of a version?" It should be easy to give a precise, informal *feature statement*. Note, however, that the feature statement must take into account the presence or absence of other features. A special case is that f cannot be added at all unless feature g is already present. This would mean that f depends on g.

5.3 Interaction Statements

Dependency is not the only reason why it may become necessary to mention other features in a feature statement. Suppose f and g are orthogonal, i.e. there are versions with f but without g and vice versa. If the functionality of both features is to apply some transformation to the same data, then the order in which these two transformations are performed may be significant. Therefore, the feature statement of f would be incomplete without specifying that f must happen, say, *before* g. Similarly, the statement of g would have to require that g happen *after* f.

The above relationship between orthogonal features is called *feature interaction*. Generally speaking, interaction between f and g means that it is impossible to deduce the exact behavior of a version in which f and g occur together from the behavior of versions in which only one of them occurs. Interactions between three or more orthogonal features are also possible.

Note that every feature interaction must clarify an ambiguity introduced by the simultaneous presence of orthogonal features. If a straight-forward combination of the features is possible, then there exists a version in which the features are all present without any interaction. Adding unsolicited interaction between the features would increase complexity, so the "interaction" is really a new feature.

Clearly, specifying an interaction in all the corresponding feature statements would introduce an awkward redundancy into the specification. Therefore, feature interactions should be expressed in *interaction statements*, separately from the feature statements. Through this convention, feature statements may only contain references to prerequisite features, not to orthogonal ones.

6 Feature-Oriented Design

6.1 Core Design

Similarly to the specification phase, in feature-oriented design we deal first with the core functionality. As feature-orientation is orthogonal to existing paradigms, any appropriate design method can be used to design the core version.

It is important to review the core design with respect to features that are already known from the specification. While some features might fit in smoothly with the anticipated system structure, others might be impossible to implement without major restructuring. The core design should be adjusted accordingly, until every known feature is expected to be implementable in a clean way.

6.2 Library Design

Another issue to be addressed during core design is to identify all the library functionality needed to implement the core functionality. Some of it will be covered by existing libraries, the rest has to be designed as new libraries and logically separated from the application design.

The decision what to delegate to a library and what to leave inside the application is often based on a tradeoff. This is true for existing libraries as well as for new ones. On the one hand, we want to move out of the application as much functionality as possible to keep it simple. On the other hand, we lose the flexibility to make arbitrary, application-specific changes once we incorporate functionality into a library.

While the distinction between genuine application modules and new libraries written during the project seems not too important in conventional software engineering, it becomes vital in feature-oriented development. The reason is that functional enhancements to application modules are always explicitly registered as features, whereas, e.g., adding an entry point to a library module is not considered a new feature.

6.3 Feature Design

The core design is *not* updated when a new feature is designed. Like the core specification, we want to keep the core design simple, so flooding it with features is not a good idea. Moreover, changing an existing document is a tedious endeavor, and it's hard to keep track of what has been changed at what time and for what purpose.

As in feature-oriented specification, we keep feature designs textually isolated from the design of the core version and from each other. Designing a feature means developing an implementation strategy that satisfies the semantics of the feature statement. It is often possible to encapsulate most of a feature in a new implementation module, which would then be designed with some existing design notation. How the new module must be "hooked" into the rest of the system, however, is explained in plain prose (*feature attachment*).

Derived Features

The designer of a feature-oriented system must always be aware that a unique correspondence between application code and features is required, i.e., features may never share code. If it turns out during design that there would be shared code, then the common functionality must be factored out into a new feature and removed from the original features. Since the new feature was not visible at specification time, we call it a *derived feature*.

The rule that features never share code may seem a little arbitrary. But one of the problems with conventional software is that code is often shared by many features, and the way in which a code fragment contributes to each of them is never made explicit. With our "pure" fragments, on the other hand, there is no such problem because each fragment contributes to one feature and nothing else.

Let us look at an extreme example. With code sharing, it would be perfectly legal to say "all the features are implemented by all the code". Clearly, this reveals nothing about the code/feature relationship.

Derived Dependencies

Every derived feature causes *derived dependencies*, because the features for which the shared code was originally needed now depend on the new feature.

But this is not the only way a derived dependency can emerge. In many cases, the easiest way to implement a feature is to reuse the functionality of some other feature, even if there is no logical dependency between them. The features may be orthogonal in the specification, but because of the design decision to base the implementation of f on g, a reduction with f but without g becomes impossible. Hence, f depends on g.

6.4 Interaction Design

Apart from core functionality and features, we also need to design every feature interaction, i.e., to plan the incorporation of code that ensures that the interaction will happen exactly as specified in the interaction statement.

Like in the specification, interactions are designed in textual isolation from the rest of the system. If a new module is introduced for the sake of a particular interaction, then the module design becomes part of the interaction design. If existing modules have to be modified (*interaction attachment*), then this is described verbally.

It sometimes happens that an interaction statement requires no code at all to be implemented. One example are interactions like "f has to happen before g". In sequential programming, putting the code for f and g in the right order will do the job. (In concurrent programming, additional code for synchronization may be needed.) Another example are interactions that fall in place automatically as a consequence of the system structure. Because there is no explicit program logic that implements such interactions, we call them *implicit interactions.*

Whether an interaction is implicit or not is unknown at specification time, because it depends on the implementation strategy. Therefore, implicit interactions must be explained explicitly in the design.

Derived Interactions

In analogy to derived features and derived dependencies, there are also *derived interactions.* If the implementation strategy of two features causes an undesired *feature interference* as soon as both come together, then additional code is needed to correct this problem. Since the implementation strategy, and therefore the interference, cannot be known at specification time, this code is a derived interaction identified during design.

Failure to detect all the feature interferences is a major source of software errors. Although we cannot offer a recipe that guarantees complete detection, we believe that an interference is more likely to be noticed in a feature-oriented design than in a traditional design. After all, we can easily draw a matrix whose rows and columns are labeled with features. If we spend some time thinking about every field in this matrix, we should be able to explain how the two features interfere, or give an argument why they don't.

7 Feature-Oriented Coding

7.1 Marked Fragments

The basic idea behind feature-oriented coding is to mark all the code fragments that are responsible for a certain feature. This may seem an expensive overhead to the coding task, but improvements in software quality do not come for free.

The quality of the code is what ultimately determines the quality of the product. Rather than emitting as many lines of code per day as possible, programmers should be encouraged to think carefully about the precise correlation between the functionality they want and the code they write. The obligation to mark code fragments enforces this kind of awareness, which in turn raises the pride of the programmer and the quality of the code.

A sophisticated environment for code marking with a syntax-directed editor can be found in [3]. In order to demonstrate that the benefits of feature-oriented coding can be achieved with much simpler tools, we use the C preprocessor cpp available on every UNIX system. The following yacc example shows a fragment that belongs to the feature where:

```
select:
  SELECT { select_clear(); }
  sel_attributes FROM sel_tables

  #ifdef Fwhere
  where_clause
  #endif Fwhere

  '\n' { select_go(); }
;
```

Nesting of marked fragments is also possible, but only if a relationship between the features has been identified during design. One case is feature interaction, because we must make sure that the interaction code disappears as soon as one of the associated features disappears.

The second case is feature dependency, although strictly speaking nesting could be avoided there because the dependent feature may never be enabled without the prerequisite feature. However, it is often more natural to nest the code of the dependent feature inside the prerequisite feature. Nesting a prerequisite feature inside a dependent feature is, of course, forbidden.

7.2 Code Suppression

Although code insertion is the cleanest way to attach a feature, it is sometimes necessary to modify existing code. This can always be achieved by suppressing old code and inserting new code. Using cpp directives, we can write:

```
#ifdef Fxxx
<...code with xxx...>
#else  Fxxx
<...code without xxx...>
#endif Fxxx
```

However, this option should be used with extreme caution and only if there is no similarity between the two fragments. Otherwise, each new feature would cause code duplication, which would blow up the code and introduce a lot of redundancy. Therefore, suppressed code fragments must be made as small as possible and should always be annotated with a comment that explains why the suppression was unavoidable.

Most of the time, there is a way to do without suppression. Sometimes the existing code has to be slightly rearranged, e.g., by introducing a new auxiliary variable, but this is only good for the clarity of the code. In our personal experience, so far we have always been able to do without code suppression.

7.3 Version Generation

The benefits of marked fragments for code documentation and programming discipline have been mentioned above. Another advantage is the ability to generate reduced versions with a simple preprocessing step. Actually, this is the main advantage on which all the other advantages rely. The programmer is not only responsible for the correctness of the full version, but also for the correctness of every reduction. This makes coding much more challenging, and it fosters exactly the kind of alertness on which software quality depends.

8 Feature-Oriented Testing

A feature-oriented implementation contains much more information than a conventional implementation. In addition to the full version, every possible reduction is implied by the implementation and can be built automatically by feature preprocessing. Feature-oriented testing uses this additional information in order to obtain more significant test results.

8.1 Feature-Oriented Correctness

For a feature-oriented implementation to be correct, not only the full version must be correct, but also every reduction. The correctness of a reduction can be tested by comparing its behavior with the corresponding reduced specification. If the full version seems correct but some reduction behaves incorrectly, then this indicates that the relationship between code and functionality has not been completely understood by the programmer.

This means not only that the functionality is incorrectly documented in the source files, but also indicates an increased probability that there are other problems with the code. It may even be that there is an error in the full version that would have never been found by testing the full version alone.

Another advantage of feature-oriented testing is that the testing team can become familiar with the system functionality gradually. They can start testing the core version, and if they don't find any errors, they can add a feature and test it against the specification. To test a feature interaction, they would activate all the features that participate in the interaction. Of course, the correctness of extremely reduced versions is not a *sufficient* condition for overall correctness. But it is a *necessary* condition, and it allows the testers to concentrate on one feature at a time.

8.2 Extent of Testing

For very big systems, it may be expensive to test every single version. On the other hand, the more resources have been allotted to testing, the more versions can be tested. The most profitable versions are probably the ones close to the full version and close to the core version.

What should be done for every version, though, is to see if it compiles. It is straightforward to write a tool that reads all the features and dependencies, enumerates the feasible feature subsets, and builds all versions. Even though this is only a *necessary* condition for correctness, problems such as misuses of constants, record fields, variables, or routines outside the intended feature are all detected at compile time. Similarly, if the compiler reports unused variables or record fields, then the programmer might have forgotten to put them in a properly marked fragment.

If automated testing is desired, we could even go a step further and write "feature-oriented" test cases with marked fragments. An acceptance procedure would also be written with marked fragments, so that a complete automatic test could be built and run for every version.

9 Feature-Oriented Maintenance

The goal of feature-oriented maintenance is to transform one consistent feature-oriented implementation into another consistent feature-oriented implementation. It cannot be overemphasized that a *transformation* and a *feature addition* are different activities. Adding a feature always means a transformation, but not every transformation is due to a feature. Marked fragments are only used for features, not for other transformations.

There are four typical kinds of transformations:

- functional enhancement
- functional change
- error correction
- internal restructuring

9.1 Functional Enhancement

Functional enhancement is the most natural kind of transformation in a feature-oriented implementation. However, it is important to break down enhancements into features of reasonable granularity, and to mark new code fragments according to the feature to which they belong.

Ideally, a feature can be implemented by mere insertion of new code (and maybe code suppression). In this case, the feature-oriented correctness of the reduced version comes for free, because as soon as we disable the new code fragments, we have exactly the system as it was before.

If slight rearrangements have to be made in order to implement the feature in a clean way, then some regression testing is required. For regression testing, the new feature is disabled, and the system is expected to behave exactly like it did before the transformation. Automatic testing can be of great help here.

In the worst case, the enhancement is impossible without major modifications to the internal system structure. In this case, a separate restructuring transformation must be applied first, as described below.

9.2 Functional Change

A change to the system functionality does not always increase complexity. Requirements include many choices the customer might as well have made differently, and if such a choice turns out to be detrimental, it will be replaced by a different one, typically without a change in complexity.

A functional change might even reduce complexity, typically if the customer finds out that a certain anomaly or restriction that was put in the initial specification was not such a good idea after all. A special case occurs when an entire feature is not wanted any more. Then there is no need to touch the code, because the feature can be disabled. Notice, however, that every dependent feature will be disabled, too.

In general, changes to existing parts of the specification, the design, and the code will have to be made in order to realize a functional change. A good rule of thumb is that everything should look afterwards as though no change had ever been made (except for a change log in the source). We don't want to bother future maintenance engineers with facts that are no longer current. Their "historical" understanding of the system should be restricted to reduced versions of the current implementation, even if this is not the true history. (The idea to "fake" history stems from [7].)

9.3 Error Correction

Error corrections are handled much like functional changes. Depending on the phase in which the error occurred, we must update the specification, the design, or just the code. Under no circumstances should a correction be treated as a feature, e.g. by marking the fragments that correct the error. A proper entry into the change log is, of course, indicated, but we will never "reduce" to the erroneous version unless it is a simplification.

If the erroneous version *is* a simplification, however, then "correcting" the error is really adding a new feature. Therefore it should be treated as a feature to begin with.

9.4 Internal Restructuring

The last kind of transformation, internal restructuring, is supposed to have no impact on the system behavior. (The only exception is that error corrections are allowed if the errors disappear as a side-effect of the restructuring.)

The reason for restructuring is to facilitate some functional change or enhancement that would otherwise be awkward to implement. Ideally, the initial design anticipates many future changes, but features that are requested many years later are often impossible to predict.

In traditional maintenance, restructuring usually happens "on the fly", together with major modifications of the system behavior. With this approach, it is anything but easy to verify whether both the restructuring and the functional changes have been done right. Automatic regression testing against old test data tends to be difficult because of the changed system behavior.

A cleaner approach is to separate structural and functional changes, starting with the necessary restructuring. Since the claim is that the system behavior has not changed as a result of restructuring, regression tests with old test data are possible in a trivial way. A correctly restructured version with the old behavior can then be used as the basis for further functional enhancements.

The separation of structural and functional changes is not limited to feature-oriented systems but also works for conventional implementations. However, feature orientation greatly amplifies its benefits in two ways: First, a feature-oriented regression test over all the reduced versions is much more significant than a regression test of the full version alone. Second, the regression test is not confined to the moment immediately after restructuring, but is still possible after many new features have been added. By disabling all the new features, we can easily test whether the major restructuring plus all the subsequent minor rearrangements have preserved the original behavior.

10 Tool Support

It seems that the full potential of feature-orientation has yet to be tapped. The gains in software understandability, testability, and maintainability look very promising, even if only simple tools are available and features are only controlled at compile time.

More experience with larger systems will lead the way to more powerful tools, which could, for instance, keep track of feature dependencies and interactions in order to perform consistency checks on the implementation. To cope with large numbers of atomic features, support of hierarchical structuring will also become necessary.

As we gain more experience with feature orientation, we also expect it to feed back into programming languages and compilers. Current restrictions occasionally cause difficulties in code marking, e.g., because no comma may follow the last item of a comma-separated list.

Although marked fragments are meant to make the code more transparent, excessive marking can easily reduce readability. Of course, there is no royal road to understanding complex software, but we do believe that there are better ways of presenting marked fragments to the human eye. For instance, feature-oriented editors could enclose code fragments in boxes, or distinguish features by colors. It would also be nice if the user could control the visibility of features with simple editor commands, thus walking through horizontal abstractions and seeing only what he wants to see at any moment.

11 Formal Specifications

In the past, a frequent objection to formal specifications has been that defining every detail formally is too much work and only leads to complicated, unreadable specifications. With feature orientation, however, there are some new alternatives. For instance, we could use formal methods to specify and develop a core version, but specify additional features informally. This would reduce the complexity of the formal paperwork significantly and thus increase our confidence in the core version.

Another possibility is to go a step further and include also some fundamental features in the formal specification. An interesting option is to mark fragments of the formal specification according to the features they stand for. This would make the formal specification more transparent, and the semantics of every feature would be captured formally.

References

[1] H. Egghart and E. Knapp. A feature-oriented approach to high-quality software. In *Proceedings of the Fourth International Conference on Software Quality*, McLean, VA, 1994. To appear.

[2] H. Egghart and E. Knapp. A feature-oriented software life-cycle. In *Twelfth Annual Pacific Northwest Software Quality Conference*, Portland, OR, Oct. 1994. to appear.

[3] C. F. Hart and J. J. Shilling. An environment for documenting software features. *ACM Sigsoft*, 15(6):120–132, Dec. 1990.

[4] D. Hough. Rapid delivery: An evolutionary approach for application development. *IBM Systems Journal*, 32(3):397–419, 1993.

[5] B. J. Keller and R. E. Nance. Abstraction refinement: A model of software evolution. *Software Maintenance: Research and Practice*, 5:123–145, 1993.

[6] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–137, Mar. 1979.

[7] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, Feb. 1986.

[8] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings IEEE Conference on Software Maintenance*, pages 200–205, Orlando, Nov. 1992.

Requirements Monitoring for System Maintenance in Dynamic Environments

Stephen Fickas
Department of Computer Science
University of Oregon, Eugene OR 97403
email: fickas@cs.uoregon.edu

Martin S. Feather
USC / Information Sciences Institute
4676 Admiralty Way, Marina del Rey CA 90292
email: feather@isi.edu

Abstract

We propose requirements monitoring to aid in the maintenance of systems that reside in dynamic environments.

By requirements monitoring we mean the insertion of code into a running system to gather information from which it can be determined whether, and to what degree, that running system is meeting its requirements. Monitoring is a commonly applied technique in support of performance tuning, but the focus therein is primarily on computational performance requirements in short runs of systems. We wish to address systems that operate in a long-lived, ongoing fashion in non-scientific, enterprise applications.

The results of requirements monitoring benefit the designers, maintainers and users of a system - alerting them when the system is being used in an environment for which it was not designed, and giving them the information they need to direct their redesign of the system.

Our approach builds the relationship between (decompositions of) the system's idealized requirements, assumptions made of the system's environment (these assumptions are what get monitored), and remedial actions to be taken to evolve the system design when those assumptions are violated. This approach is illustrated on a license manager operating in a distributed network setting.

1: Requirements monitoring in dynamic environments

We focus on requirements engineering issues arising in domains where the environment cannot be counted on to remain static. The general problem is that requirements, and the designs that emerge from those requirements, are typically formulated within the context of an assumed a set of resource and operating needs and capabilities. As the environment changes, it may render those assumptions invalid, necessitating the corresponding evolution of the system. This phenomenon is particularly prone to occur in what Lehman has termed "E type systems", whose installation in some real world domain induces changes in the environment itself, and so leads to altering the system's own requirements [Lehman 1980].

The two major questions we have been studying are as follows:

i) How can we know when our system needs to be evolved? In particular, how can we carry through to run time the assumptions of resource and operating needs and capabilities made at design time? For example, if we design our system under one set of assumptions about the environment, how can we know when they become invalid once the system is in operation?

ii) Suppose we could detect environment changes that necessitate evolution of our system - how can we use this information to orchestrate this evolution? In the ideal case, we would like this process to be automatic: monitoring information would be consumed by the system itself, which would adjust its own structure or functionality. More prosaically, we may supply this information to the human maintainers of the system, who will thus be aided in their task of evolving their system.

Our approach has been to cast the first question as a problem of *requirements monitoring:* we advocate that as part of the design of a system, requirements monitors be installed to gather and analyze pertinent information about the system's run-time environment. We formulate specifications of what to monitor so as to gather the information needed to detect divergences from our assumptions that adversely affect adherence to requirements. We address the second question by recording at design time not only the requirements, but also the assumptions comprising the context in which those requirements were formulated, and the compensatory evolutions that we might employ when those assumptions become invalid.

2: An example - license managers

We have studied the above issues using a small but representative problem, a distributed license manager running in an enterprise. The purpose of a license manager is to allow duplicate copies of a piece of software to be used simultaneously by some number of users; the enterprise will have purchased a number of 'licenses' for that software, and at any one time up to that many users should be allowed to simultaneously use the software; the vendor, from whom the licenses were purchased, relies upon the license manager to ensure that at any one time the number of simultaneous users does not exceed the number of purchased licences.

License managers are of interest to us because there is a wide range of environments in which they reside. The environmental features affecting the license manager include number of potential users, patterns of usage, number of licences purchased, network performance, and available computational resources. More importantly, those environments are often *dynamic* - that is, their features vary over time. The number of potential users may vary, new computational resources replace old ones, etc. Such volatility is inevitable in companies that keep pace with changing economic circumstances and changing technology.

We have studied one license manager in particular - FlexLM, distributed as part of the Solaris software package by Sun. FlexLM has been designed to be applicable in a wide variety of environments. It offers a set of "design parameters" to tune. In essence, the designers of FlexLM anticipated a range of different types of environments in which the system might be deployed, and provided system administrators with some design freedom in setting up the license manager for operation in their particular environments. Thus for our purposes, the 'design task' we focus upon is the selection (and re-selection) of those parameter settings. This frees us from the need to modify program code, which is itself a very difficult task!

We are interested in what happens as changes occur to the environment in which the license manager has been placed. An example is a change in pattern of usage by users - if they become tardy in returning licenses when they no longer need them, and this is causing other users to have to wait a long time to get a license, the administrator may wish to switch to a design in which licenses have a time-bound placed on them (or to decrease the time bound if such a design is already in use).

We will return to the example of license managers after describing more of our general approach.

3: General approach - linking requirements, assumptions and evolutions

In general, our approach is to establish the relationships among the following three concepts:

- the overall *requirements*,
- the *assumptions* made about the current state of the environment, and
- the set of remedial *evolutions* available when mismatches develop between assumptions and the current environment.

We propose the use of monitoring to detect the relevant changes to the system's environment. What to monitor for is determined by consideration of the relationships among requirements and assumptions. This yields a specification of monitoring needs. From such a specification, run-time monitoring code (that gathers information and performs analysis) is compiled. Related work on monitoring for debugging and performance tuning provides existing capabilities for such compilation (for a survey of such work, see [Mansouri-Samani & Sloman, 1993]). Note that for debugging or performance tuning the perturbation caused by the insertion of monitoring code threatens to disturb the information gathered and the conclusions drawn from that gathered information, and so must be done with great care. However, what to monitor for is usually obvious, for example, unbalanced loads on multiple processors, or communication bottlenecks. In contrast, for our purposes the perturbation induced by insertion of monitoring is not usually of great concern because it will not usually alter the inferences we draw from the gathered information, whereas the determination of what to monitor is the essence of the problem. Thus we feel confident that once we have developed monitoring specifications (i.e., determined what to monitor for) we can readily apply existing monitoring tools and techniques to create the actual monitoring code. We therefore will focus solely upon the relationships between requirements and assumptions about the environment, to understand how they give rise to monitoring specifications and how the results of monitoring can be applied.

4: More on the license manager example

The overall requirements of most license managers are as follows:

1. *At any one time, the number of simultaneous users of a piece of software should not exceed the number of licenses purchased for that software.*
2. *Users should not have to wait unduly long for a license.*
3. *No more licenses than are necessary should be purchased.*
4. *Users should find the license manager to be as unobtrusive as possible.*
5. *The running license manager itself should not overly burden the system resources (cpu time, network bandwidth, storage space).*

Note that the vendors of software, and the users of software, have competing interests. For example, users might like to 'cheat' by violating the first requirement, while vendors might prefer a license manager that made it likely that users would purchase more licenses than strictly necessary. The license manager itself sits in the middle of these competing interests, and our presumption is that the above set of requirements (or something like it) represents a balance deemed fair and acceptable to all concerned.

In most cases, requirement 1 is a 'hard' requirement, ensured by the license manager. There is little purpose in trying to monitor for violations to this requirement, since our monitoring would likely not be any more effective than the license manager itself in detecting violations.

The remaining requirements are typical of 'soft' requirements, which are tricky to design for, particularly in the context of a dynamic environment. They are expressed with varying degrees of precision (e.g., numbers 2, 4 and 5 are stated rather informally). They may be mutually incompatible (e.g., improved satisfaction of 2 through 4 may require consumption of more system resources, thus degrading satisfaction of 5). It is these requirements that induce the greatest need for the kind of monitoring we advocate, and offer the greatest challenge to determine precisely what to monitor for. Because FlexLM's design parameters give us the freedom to tune its installation, we can readily explore a large space of alternative designs that achieve a variety of compromises among these requirements.

To illustrate our approach, we now consider one of the requirements in more detail: 2. *Users should not have to wait unduly long for a license to use a piece of software.* We manually subdivide this requirement into several cases, each of which is a finer-grained requirement. For each subdivided requirement we identify the corresponding assumption(s), and in turn, for each assumption, the corresponding remedy(ies) of how to evolve the design in the case that the assumption is violated - table 1.

In general, subdivision of requirements is done by following a process closely related to that described in [Dardenne, van Lamsweerde & Fickas, 1993]. A top-level requirement is subdivided and the assumptions behind the resulting sub-requirements are identified, to emerge with assumptions that are candidates for monitoring and remedial action. Generally, this process clarifies the informality present in

Table 1: *Users should not have to wait unduly long for a license*

Subdivided Requirement	Assumption	Remedy
Licenses sufficient for user population	User population < k	Purchase more licenses or reduce user population
	No more than x% of user population wants to use at once	Purchase more licenses or reduce user population
Individual users served licenses fairly	Longest waiting user gets license first	Have license manager maintain queue of waiting users
	Users do not hog licenses	Issue time-bounded licenses
		Revoke licenses of current users
Users not kept waiting if licenses are available	License manager on reliable platform	Relocate license manager to more reliable platform
		Employ more robust license manager design (backup, majority)
	License requests do not become backlogged at license manager	Subdivide license manager & licenses across several platforms

the initial requirements. The last step is to identify possible remedies to apply when the assumptions are violated; remedies take the form of evolutions to be applied to the system's design.

For example, the initial requirement can be monitored (by watching for a user who is kept waiting longer than some pre-determined time for a license), but has no immediately identifiable assumptions or remedies to take upon detection of violations. In contrast, the above sub-requirements do have clear assumptions underlying them, such as the bound on the user population. Some of the remedies are straightforward, although not necessarily acceptable (e.g., purchase of more licenses will require additional funds, which might not be available). Some depend upon conditions that arise because of the imperfect nature typical of the distributed environments within which most license managers must operate - communication over networks can degrade or fail, individual machines (on which users and/or the license manager itself are running) can become overloaded or fail. For example, we may make an assumption of high reliability of the machine on which the license manager will be located, and, on the basis of this assumption, select a design that will (i) cause the license manager to run on that one machine, and (ii) cause licenses to expire whenever the license manager itself is inoperative (in particular, when it's machine crashes)[1]. Monitoring for violations of this assumption (i.e., downtime of the machine on which the license manager is located) can be used to detect when this has caused users holding licenses to lose the use of them, and waiting users to be unable to get a license. One possible remedy is to switch to a design in which the license manager is replicated across several machines, and a user's license remains valid as long as that user remains in live communication with a majority of those machines. Note,

1. The latter may seem to indicative of a poor design decision, but in fact admits designs in which licenses are quickly 'retrieved' when users' machines crash, and so supports requirement 4.

however, that this design is less satisfactory with respect to requirement 5, and so should not be selected without good reason.

As well as gathering information on how the assumptions and requirements are met (or not met) by the current design, monitoring can also be used to answer 'what if' questions about candidate alternative designs. Continuing the preceding example, if the current design is of the manager running on a single machine, we could monitor for how much more reliable it would have been had the manager been subdivided into several incarnations running on separate machines, by monitoring the status of not only the license manager's machine, but also the status of those other machines.

5: Current status

Currently, we do the subdivision of requirements and identification of assumptions and remedies manually. We have experimented with monitoring a *simulation* of license management, modeling the key concepts of users, licences, etc., and encoding monitoring queries as AI-like daemons that watch for occurrence of those monitoring conditions. This is straightforward to do using our in-house AP5 environment, which provides modeling capabilities together with the ability to declare daemons whose triggers have access to all the information present in the model [Cohen 1989]. Our focus has been the determination of what to *monitor* for; making the monitoring itself efficient has previously been studied by our colleagues [Liao, 1994]. Our next step is to monitor the functioning of FlexLM itself in a distributed environment. For this purpose, we will be using standard SNMP management stations as a means of monitoring our queries in the network setting within which FlexLM operates.

6: Observations

License managers are, we think, are a small but representative example of systems that must operate in dynamic environments. Its 'soft' requirements are challenging to balance in the context of its operating environment, and its design must necessarily make assumptions about that environment. Monitoring shows promise as the means to determine when those assumptions are become violated, and whether, as a consequence, its requirements are not being met. It is particularly interesting to note that although the initial expression of requirements often lacks formality, requirements in conjunction with assumptions readily suggest easily formalized monitoring specifications.

In conclusion, we believe a focus upon requirements of systems that operate in dynamic environments suggests the need for monitoring as a means to guide the appropriate evolution of those systems.

7: References

[Cohen, 1989] D. Cohen. Compiling complex database transition triggers. In *Proceedings, ACM SIGMOD International Conference on the Management of Data, Portland, Oregon.* SIGMOD RECORD 18(2), June 1989.

[Dardenne, van Lamsweerde & Fickas, 1993] A. Dardenne, A. van Lamsweerde and S. Fickas. Goal-directed Requirements Acquisition. *Science of Computer Programming,* 20(1-2):3-50, 1993.

[Lehman, 1980] M.M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. In *Proceedings of the IEEE,* 68: 1060-1076, 1980.

[Liao, 1994] Y. Liao. Efficiently Computing Derived Performance Data. *Automated Software Engineering,* 1(1):11-30, 1994.

[Mansouri-Samani & Sloman, 1993] M. Mansouri-Samani and M. Sloman. Monitoring Distributed Systems (A Survey). Imperial College Research Report No. DOC92/23, Imperial College, Dept. of Computing, 180 Queen's Gate, London SW7 2BZ, UK (revised version published as Chapter 12 of *Network and Distributed Systems Management,* M. Sloman (ed.), Addison Wesley, 1994: 303-347.

Formal Methods Experience and Recommendations

Formal Methods Group
Department of Defense
Fort George G. Mead

Abstract: *The Federal Government has supported the development of Formal Methods for at least the last 20 years. Indeed, it has been the leading funder of Formal Methods research and technology during this period. This paper discusses our current Formal Methods program while highlighting our experience and making recommendations for change in our emphasis.*

1.0 Introduction

The Federal Government has leant substantial support over the last 20 years to the development of tools that use mathematical formalisms to aid in the specification and verification of computer systems. The use of mathematics to describe computer systems has gone under various titles. Among them are "the mathematics of software engineering" and "Formal Methods". For purposes of the present discussion, the term "Formal Methods" shall be used.

The decision to invest heavily in Formal Methods was made because of the increasing realization that while the testing of computer systems found errors that could be corrected, sometimes at great cost, that testing did not guaranteed that all errors would be found and corrected. Arguments were made by researchers in the Formal Methods field that only the mathematical idea of formal deductive proof could ever guarantee that software functioned correctly. We began to support the development of software tools that implemented proof procedures of mathematical logic. We call these tools software proof systems. Over the years, we have supported both directly and indirectly the construction of many of the software proof systems currently in existence.

Advanced evaluation criteria have required formal verification that the top-level design implemented the requirements specification. Placing the formal verification requirements at the top-level is a good idea for two reasons: (1) the state of the are then (and now, and for the foreseeable future) did not provide the ability to verify large amounts of implementation software; and more importantly, (2) it was understood at the time that the major problems with software are usually traceable to design flaws. This is still the case today, and provides the basis for our later arguments that we should place our Formal Methods emphasis on specification tools for the design level.

2.0 Experience

Our Formal Methods activity has not, to date, been a practical success. This is despite the fact that there has been a dramatic improvement in the understanding and use of software proof systems, and some notable successes in specialized areas. There are several reasons for this lack of success:

- Proving theorems is a difficult and challenging process.
- The software proof systems developed to date are difficult to use, in part because they have poor human-machine interfaces; in part because they require a deep understanding of proof techniques, which are only partially supported by the proof system; in part

because straight-forward but tedious proof elements have yet to be fully automated; and in part because it is difficult to formulate practical problems in precise mathematical statements.

- University training in the US does not prepare engineers and computer scientists to use mathematical proof techniques.

In hindsight, it is clear that the technical thrust of our program in Formal Methods has placed too much emphasis on mathematical proof, and too little emphasis on mathematical specification. Had we placed more emphasis on training engineers to use the mathematics of software engineering to specify systems, we might have been able to make better use of mathematical proof in particular instances.

3.0 The Importance of Specification

We have noticed that proof is effective in specialized areas, while specification is effective as a language of system description. This view forms the basis for our belief that Formal Methods be considered in a broader context than it has in the past, and that we place increasing emphasis on using mathematics to specify systems. It is likely that the use of software proof systems will always require a specialist grounded in mathematics, logic, and computer science, but those specialists must always depend on the specifications from a broad community of designers and evaluators in order to do useful work. We are now in a position to use the Z specification language effectively, and as a result should make the recommendation that we must emphasize the use of formal specification to aid our goal of developing correct system implementations.

4.0 Application Areas

There are five major Formal Methods application areas in which our research and technology can make a significant contribution. They are:

1. Formal modeling and analysis of trusted distributed systems.
2. Hardware technology, specifically the verification of microelectronic devices.[1]
3. Human-machine interfaces for Formal Methods tools.
4. Training programs for a broad community of potential users of Formal Methods, which first emphasize the reading, and then the writing of formal specifications.
5. Formal Methods standards.

4.1 Trusted Distributed Kernels and Operating Systems

Fundamental to a modular approach to correct systems is the understanding and development of operating systems (resource managers) that support secure applications. The evolution of operating systems has led to a layered model of system functionality. User applications form the top layer, progressing next to sub-systems such as a database system. At the bottom layer, sitting on the hardware is a core module that supports activities like memory management, process activation, and interprocess communication. The words "kernel" and "micro-kernel" have become words of choice to describe these core resources. There are other variants, for example "separation kernel" and "virtual-machine monitor", which imply some special feature. Unless we are directly referring to a special feature, we shall use "kernel". The appropriately designed kernel can sup-

1 We consider hardware issues to be an important part of the overall Formal Methods picture. However, they fall outside the remit of this workshop, and are not discussed in this document.

port modular applications ranging from file system to name servers to database systems. The fundamental objective of our research program in distributed systems is to develop techniques that enable kernel-based systems to operated correctly. This requires an understanding of computer security models, kernel models and distributed computing models, and there has been a great deal of research into these issues over the last 20 years.

The use of Formal Methods to specify systems has played an important role in the development of correct systems, and should continue to do so. Our major application activity should be to support the development of trusted distributed computing environments. In the short term, software proof systems will not be effective in gaining assurance of the correct functioning of kernel-based systems. However, short term assurance can be achieved in part by using a formal specification of kernel-based systems to develop a suite of test programs. Such a suite must provide adequate coverage of day-to-day distributed system operations.

4.2 Human-Machine Interfaces

Current human-machine interfaces for Formal Methods tools have limited utility. Work of any consequence to improve this state of affairs is just now beginning. Because we have a broad experience over the range of Formal Methods tools, we have special opportunity to develop high quality interfaces that can be used across the spectrum of tools. We have already begun work on interfaces for several systems.

There is more to the interface question than just human-machine interaction. The more we apply Formal Methods to real (or realistic) problems, the more we will need to interact with existing design tools and methodologies. An important interface issue (perhaps, more important than human-machine interfaces) is between Formal Methods tools and existing components of the design process. Potential customers of Formal Methods technology are not interested in the capabilities of Formal Methods in isolation. They want to know how to the technology fits with what they are doing now. For example, in the hardware design are, engineers will ask: Can a formal specification language and a software engineering design tool work together in some way? Can a subset of VHDL formally specified in a software proof system match up with the actual use of VHDL in a microelectronic device? How can one translate a microelectronic circuit netlist into something understandable in a software proof system? Is Formal Methods meant to supplant certain steps in the design or analysis flow, or is it meant to exists in parallel with current processes? These are challenging issues, and leads to our view that we must develop methods and procedures to integrate Formal Methods with software tools that form the basis of our design process.

4.3 Education and Training

In order to support all of these recommendations, there must be a greatly expanded training program to introduce our entire organization to formal specifications. We need to develop the capability to teach specification languages to any engineer, computer scientist, analyst, or manager who wants to understand them. The training must be quick (a couple of weeks at most), and it must enable the students to read formal sections of documents.

4.4 Standards Activities

International standards organizations are beginning to standardize Formal Methods languages. We are participating in some of these standards activities, which provide excellent opportunities to understand and affect the direction of these standards. Our experience in human-machine interfaces, specification and software proof systems should help in achieving standards that will broaden the community that can use Formal Methods.

5.0 Conclusions

The Federal Government has been in a unique position to observe and influence the development of Formal Methods, and we have come to the conclusion that the way forward is to move away from the narrow view that the primary contribution of the field is the development and use of software proof systems. This broader understanding should encompass more of the overall system development process from specification to testing. Our current focus is to enhance our ability to write useful formal specifications, and to apply that knowledge to real system development problems.

ANALYTICAL DEVELOPMENT OF CONTROL-INTENSIVE SOFTWARE

R. Hardin, Z. Har'El, R. Kurshan
AT&T Bell Laboratories
Murray Hill, NJ 07974
email: k@research.att.com

ABSTRACT

We describe a way to develop and implement control-intensive software systems such as communication protocols so they are logically sound and meet stated requirements. Our methodology employs a software system called COSPAN to facilitate logical testing (in contrast to simulation or system execution testing). Logical testing is carried out on a succession of models. Starting with a high-level model (e.g., a formal abstraction of a protocol standard), successively more refined (detailed) models are created. This succession ends with a low-level model which is in fact the code which runs the ultimate implementation of the protocol. Tests of successive models are defined not by test vectors, but by user-defined behavioral requirements appropriate to the given level of abstraction. Testing a high-level design permits early detection and correction of design errors. Successive refinement is carried out in a fashion which guarantees that properties proved at one level of abstraction hold in all successive levels of abstraction.

1. Introduction

Speaking (very) loosely, the overall objectives of a software development project could be stated as follows:

TO DEVELOP:

- A Reliable Product
- Fast (in absolute days)
- Efficiently (in terms of allocated staff)
- Supportable
- Well-documented
- Robust Design (insensitive to changes in environment behavior)
- Portable Design (implementable in various settings)

In pursuit of these objectives, one may apply a variety of "techniques" with familiar names:

TECHNIQUES:

- **Rapid Prototyping**
- **Design For Testability**
- **Modularity**
- **Hierarchical Development**
- **Structured Team Management**
- **Design For Reusability**

While these development goals and the techniques to obtain them mean different things to different people, system designers and developers are increasingly aware of the value of formal techniques for defining and achieving such goals. Let us focus on communication protocols. Not only is the international community working on standards for protocol specification and testing based upon formal techniques, but many companies have developed their own formal techniques, both for particular projects and for general use. Within the scope of a formal technique, many of these terms assume specific, focused meanings, concomitant with good predictability and reliability of the methodology.

The advantage of definition by itself, however, is of little more than academic value. To be beneficial, formalism must be applied to some specific advantage. Perhaps the most obvious application of formalism involves product reliability. Once a protocol design and requirements have been formalized, it becomes theoretically possible to determine (in a mathematical sense) whether a particular implementation meets these requirements. This is known as formal verification. The value of a particular formal verification framework may be measured by the scope or generality of the requirements for which an implementation may be tested.

For example, a framework in which only "safety" properties may be verified (properties of the form "such-and-such bad event cannot occur"), may be inadequate for analyzing communication protocols, where a paramount concern is, for instance, that once a message is transmitted, it is eventually received (a non-"safety" property).

Even if the immediate objective is only to "certify" that a given implementation meets stated standard requirements, presumably one needs to verify as well that the stated standard requirements are enough to guarantee the overall good behavior of the protocol. Unfortunately, protocol behavioral requirements extend beyond those associated with certification. In order to ensure the proper behavior of a particular implementation, extensive behavioral verification must be conducted.

It is now generally accepted that simulation or system execution is far from adequate to ensure the proper behavior of an implementation. Therefore, formal verification should have the power to draw firm conclusions about the general behavior of a system under all possible situations.

Because protocol requirements address broad properties of a protocol, formal verification is most easily (and most often) applied not to the implementation, but to a high-level model or abstraction of the implementation. It is assumed implicitly that an implementation will be true to the verified high-level model. There is a similar relationship for "certification" (the test of an implementation against a standard). In this case the standard is a high-level model; ideally certification would constitute a proof that the implementation is a correct realization of the standard, or equivalently, that the standard is a correct abstraction of the implementation. Thus, for certification of an implementation or verification of a high-level model to guarantee the correctness of the implementation, there must exist a formal association or transformation from the high-level model or standard to the low-level implementation. Let us examine this more closely.

When a protocol is defined by a high-level model that has been verified to satisfy certain requirements, it is common to say "the protocol has been verified". The next step is to implement the verified protocol model. If the implemented protocol then fails on account of an implementation decision (i.e., a construct in the implementation which is not described precisely in the model) or on account of a misinterpretation of the model, it is common to attribute the failure not to the "protocol" (i.e., the protocol model) but to the implementation. Given the preponderance of failures caused in practice by such "implementation decisions" and translations of the model, it would seem to make little difference, however, where the blame is placed. The best methodology for formally verifying a high-level model has limited value if there is no formal procedure to derive a faithful implementation from it. Often there is not even a formal basis upon which to decide if an implementation implements a model. This can be a particular problem when it masks the fact that no implementation of the protocol, precisely as it is modelled, is possible in a given environment. Such a difficulty arises not all that uncommonly when a protocol model contains implicit assumptions about environment interfaces which are not met by the given environment. If so, any "implementation" of the protocol model in the given environment is necessarily untrue to the model, and thus properties verified in the model may not hold for the implementation. Likewise, if a certification scheme involves tracking an implementation with a high-level standard, in order for this to give real information about the implementation, a formal connection is needed between the standard and the implementation. If there is a formal connection such as an association of states and transitions then proper certification should demonstrate that every transition of the implementation corresponds to the associated transition in the standard.

Even in the absence of a formal transformation from protocol model to implementation, formal analysis of the model can reveal faults in the protocol concept. However, in the absence of such faults, it may be deceptive to refer to an implementation as "verified" if there is no formal, testable relationship between the verified model and its implementation. Likewise, a certification procedure which tracks an implementation with a standard can be very useful to the extent that it uncovers discrepancies between the standard and the implementation. Without such discrepancies, however, it may be deceptive to refer to an implementation as "certified" if there is no formal, testable relationship between the standard and the implementation, or if only a few transitions of the implementation have been tested.

A simple way to define a formal relationship between a high-level model or standard and an implementation is to associate a state in the model or standard with a set of states of the implementation. Such an association, for example, may require correspondence between the receiver-ready state of the high-level model and the set of implementation states for which a certain state machine component of the implementation is in its receiver-ready state. However, since the set of implementation states for which this state machine component is in its receiver-ready state is determined by all the possible respective values of pointers, buffers, counters and so on which may occur with it, this set of states probably is very large.

Suppose that, according to the high-level model or standard, the correct transition from receiver-ready is to the state transmit. It may be that for certain implementation states (i.e., for certain values of pointers, buffers and so on), the implementation tracks the model or standard, while for others it does not. To certify truly that a high-level model or standard abstracts an implementation, one must demonstrate this not simply for a single implementation state and transition that corresponds to a respective high-level state and transition, but rather for every low-level state and transition. Probably, in lieu of symbolic verification methods, the best approach is the standard simulation routine which runs the implementation along side the high-level model or standard for as long as feasible (inevitably crossing high-level transitions many times). Nor is it accurate to conclude that if a high-level transition e.g., from receiver-ready to transmit correctly tracks a single low-level transition (for a certain value of buffers, pointers, ...), then in greatest likelihood, the receiver-ready to transmit transition would correctly track all other low-level transitions between the corresponding states (i.e., for all other possible respective values of buffers, pointers, ...). Indeed, it is well-known that the greatest weaknesses of an implementation arise at the "boundaries" of operation (buffer empty, buffer full, etc.) and that these boundaries can be very long and complex.

Since it is rarely feasible to address directly all possible transitions of an implementation (i.e., to address all possible values of buffers, pointers, ...), one must seek alternatives by which to conclude that an implementation is faithful to its high-level abstraction.

All this may sound very complicated, and indeed in purely numerical terms it certainly is. While a high-level model or standard may have as few as 50 or 500 states, an implementation typically has so many states that the numbers can be appreciated only by analogy. Given all the possible combined values of pointers, buffers and state machine controllers, one typical state space contains an estimated 10^{10^4} reachable states.

To understand this number, one may think of it this way: to determine if our implementation tracks a high-level model, we could, in theory, use a brute-force state-transition tracking algorithm. Suppose this algorithm were perfectly parallelizable among every human being on earth, each equipped with a Cray computer; the job could not be completed before the sun burns out.

Methods to manage the overwhelming complexity caused by the formal methods themselves must be supported. The classical method test as much as you can and then hope for the best now generally is agreed to be inadequate. More powerful methods must be found. Using mathematics we are able to reduce an apparently intractable test such as the one just described, to a relatively simple test with the provable property that the outcome of the simple test reflects conclusively upon the outcome of the apparently intractable test (were it to have been performed).

We use a high-level to low-level transformation in the form of a formal top-down development procedure based upon successive refinement. Starting with a high-level (abstract) model (e.g., a formal abstraction of a protocol standard), successively more detailed models are created through successive refinement, in a fashion which guarantees that properties verified at one level of abstraction hold in all successive levels of abstraction. This succession ends with a low-level model consisting of code which runs the ultimate implementation of the protocol.

We may contrast this with conventional development. Conventional development procedures add functionality in the course of development. As functionality is added, so are possible behaviors of the system. This precludes the possibility of a property proved early in the development cycle can be assured at the end of development. In the development methodology which we utilize, the implementation has fewer behaviors than the high-level design. This is what guarantees the inheritance of properties proved early in the design cycle.

Formal verification of quite general (in fact, arbitrary finitary-automaton-definable or ω-regular) user-specified requirements, appropriate to the given level of abstraction, is facilitated by reduction techniques. For each system requirement to be verified, we derive a reduction of the system relative to that requirement. The scope of such ω-regular requirements includes not only "safety" properties but also "eventuality" properties such as "the message eventually will be delivered".

The high-level model at the top of the development hierarchy constitutes a prototype of the system being developed. Since this model may be as abstract (and hence simple) as one likes, it may be defined quite rapidly, providing a rapid prototype of the system under development.

Partitioning the formal verification procedure according to the levels of abstraction defined by the development hierarchy constitutes design for testability that facilitates verification.

The basis of the hierarchical development is a modularity that permits successive module-by-module refinement rather than the hopeless task of globally refining a monolithic system. The modules may be designed to be small, simple components, so refinements easily can be designed, verified, and supported, as the system design evolves.

Modular design, the partition imposed by the development hierarchy and separate verification of numerous requirements (as dictated by the reduction techniques), all impose a natural separation of the development project into semi-independent tasks that can be performed by a development team according to a schedule imposed by the development hierarchy.

Another by-product of the development hierarchy is a form of reusability where the same higher-level model may be developed into a variety of lower-level models. Conventional attempts to design reusable code often are frustrated by the need to alter substantial parts of the code for different interfaces. This altering can have unpredictable consequences in the reused portions. The form of reusability proposed here replaces interface redesign with refinement, thus preserving properties of the higher-level reused model.

Formal verification buys reliability. Rapid prototyping concomitant with hierarchical verification allows error-detection very early in the design and development process, eliminating the much greater time and cost of redesign after development is complete. This advantage may decrease development time by an order of magnitude [HK90] over conventional methods. The development methodology naturally partitions development into parallel and generally independent tasks. Thus, the combined efforts of the development staff interfere less and hence are more efficient than in development projects where a change in one part of the project inevitably forces changes in many other parts.

The development hierarchy and system modularity also ease the burden of system support. That is, new features may be introduced at the appropriate level, verified and refined into the implementation without disturbing previously defined and verified system components. The same applies to design support: as a design evolves, its incremental upgrades are applied at the relevant level of the design hierarchy. This reduces the burden of testing and verification of the upgraded design, as it needs re-verification only from the level of the change, on down. As much design evolution entails changes only at the lowest level, this can greatly reduce the amount of testing needed to validate design evolution, and provides a simple way to implement merging (and testing) multiple changes, as discussed in [DLB94].

Because the development hierarchy affords views of the system at a variety of levels of detail, the system design is almost self-documenting.

Because low-level system interface details are introduced only at a low level in the development hierarchy, the resulting design tends to be robustly constructed relative to changes in the environment behavior definition. Likewise, the higher level system design tends to be portable, requiring only redefinition of the lowest (or lower) levels to implement the system in various settings.

All the foregoing are attributes of an automata-theoretic verification-driven top-down design development methodology called ***stepwise refinement*** [Ku94]. The design evolves as a succession of automaton models, each model more detailed than the previous. Each model is taken as a "specification" of the succeeding model (its "implementation"), and verification is used to check the consistency of each implementation with its respective specification. Earlier models in the design are more abstract, expressing only high-level logical attributes of the system under development, while the later models add low-level details needed for the ultimate implementation. The highest-level (most abstract) model in the refinement hierarchy, because it is abstract, may be developed more rapidly than a full system design, and may be considered a "rapid prototype" [Lu89] of the ultimate design. This prototype is logically debugged through verification (through attempting to verify its important attributes), in the process uncovering design errors at the earliest possible time in the design cycle. Uncovering design errors early has long been known to be an important accelerant for the design process.

In summary, formalism offers the potentials of formal verification, a formal relationship between a high-level model and its implementation, and management of the apparently intractable complexity inherent in the formalism itself. To use formal techniques to support the development goals stated at the start, each of these three potentials is required by the others; like the legs of a stool each is indispensable. Taken together, they can support the development goals, through a formalization of familiar techniques.

For an expanded discussion of these concepts and a case history of a protocol developed through this methodology, see [HK90]. For more on formal verification itself, see [Ku90], [Ho91], [Ha93], [Ku94].

References

[DLB94] D. A. Dampier, Luqi, and V. Berzins, Automated Merging of Software Prototypes, *J. Sys. Integration*, **4** (1994) pp. 33-49.

[Ha93] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer, 1993.

[Ho91] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[HK90] Z. Har'El and R. P. Kurshan, Software for Analytical Development of Communications Protocol, *AT&T Tech. J.*, **69** (1990) pp. 45-59.

[Ku90] R. P. Kurshan, Analysis of Discrete Event Coordination, *Lecture Notes in Computer Science (LNCS)*, **430** (1990) pp. 414-453.

[Ku94] R. P. Kurshan, *Formal Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton Univ. Press, 1994.

[Lu89] Luqi, Software Evolution Through Rapid Prototyping, *IEEE Computer*, **22** (1989), pp. 13-25.

Generating Software from Specifications *

Richard B. Kieburtz
Pacific Software Research Center
Oregon Graduate Institute
of Science & Technology
P.O. Box 91000
Portland, OR 97291-1000 USA
dick@cse.ogi.edu

August 24, 1994

This paper summarizes progress in applying formal methods not only to software specification, but also to create specification-driven software generators. The concepts described here are realized in tools supporting a new software development method called SDRR—Software Design for Reliability and Reuse [2], developed by the Pacific Software Research Center.

1 What is a specification?

A functional specification formalizes the functional requirements of a software component. We shall not address software architectures or system-level specifications here. A functional specification should be intelligible to an educated user, should be formal, and should be feasibly implementable. As there is no universal notation that has all of these properties, we advocate specification in mini-languages specific to each application domain. We call such a mini-language a *domain-specific design language* (DSDL).

A DSDL should be a declarative language that is implementable because it has a computational semantics, not because it is a programming language. Declarative languages offer the benefits of conciseness, readability, and removal of concerns about implementation. A DSDL manifests the high-level abstractions of the application domain. Some example DSDL's are:

- A pattern-oriented layout language for specifying prettyprinters.
- A message-specification language for messages in a command and control system.
- Control system diagrams specifying avionics control systems.

Using a formal language as the medium for expressing a functional specification in a specific domain supports a broad design space of feasible solutions. Giving the language a compositional (denotational) semantics allows software solutions to be synthesized from modular parts. Using the logic associated with the semantics specification language allows formal reasoning about properties common to all synthetic solutions.

*The research reported here is sponsored by the Air Force Materiel Command

2 Computable denotational semantics

Denotational semantics for programming languages are translations of syntax to functional expressions such that all constructions are deterministic and composable. Composability implies that the semantics of a syntactic construction is a function of the semantics of its component parts and of nothing else. If each of the semantic functions associated with a constructor of the abstract syntax is effectively computable, then we have a computable denotational semantics. Our tactic for making a specification language computable is to formalize its intuitive meaning in terms of a computable denotational semantics expressed in an executable meta-language.

We have designed the ADL language [7] as our preferred meta-language. ADL is an acronym for Algebraic Design Language. It adapts the notion of structure algebras from the mathematics of universal algebras to provide an unusually rich control structure without employing an explicit recursion operator. ADL is a language of total functions. It admits equational reasoning and program transformation by equational rewriting. ADL also incorporates a dual concept of coalgebras, which contribute control structures that correspond naturally to iteration.

Some structure algebras, most notably list algebras, are familiar to functional programmers. They have been used by Bird, Meertens and their students [4, 11, 12, 5] to derive programs from logical specifications by formal reasoning. In ADL, structure algebras are first-class entities that can be declared, bound to identifiers, abstracted to define independent program modules, and form the basis for ADL control operators.

With each signature of a variety of structure algebras (or coalgebras) that may be declared in ADL, there is associated logical rules that guide formal reasoning about properties of programs. Proofs are required, in many cases, to assure that evaluations terminate. Termination is requred by ADL's static semantics.

ADL is implemented on top of the SML/NJ implementation of the Standard ML programming language. The ADL translator takes advantage of the meta-programming extensions available in SML/NJ [6] to partially evaluate higher-order ADL combinator expressions, turning them into SML code. Thus, declarative specifications are written in a domain-specific language, which is given semantics in terms of ADL, which is in turn implemented by translation into SML.

SML provides a path to rapid prototyping of the specification. However, SML may not be the desired implementation language, thus further translation is required. Furthermore, an implementation synthesized by a direct translation of a denotational semantics can be expected to suffer from poor performance.

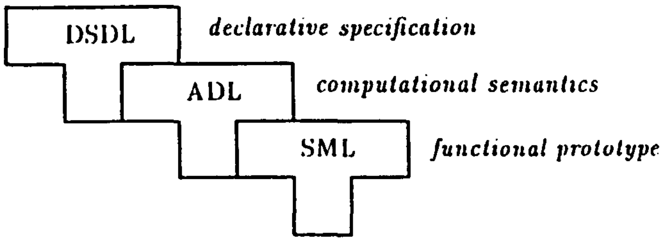


Figure 1—Translation of a declarative specification

3 Transformational improvement

The translation process generates a prototype implementation of a software component specification in terms of a higher-order functional program. This implementation is composed from small, modular program segments defined in the semantics of a DSDL. Such compositions use unnecessary intermediate data structures and duplicate control paths. Program improvement is a task for automated program transformation. We emphasize the word "automated" because human intuition is not useful to guide multi-step transformations of realistic sized programs. After one or two transformation steps, the program becomes unintelligible to a human reader.

The algebraic structure imposed on the semantic definition of a DSDL by its expression in ADL provides great leverage for a transformation system. We distinguish four categories of transformations:

(i) generic transformations that depend only upon the variety of algebras that structures a computation, but not on properties of a specific algebra;

(ii) order-reduction transformations that eliminate uses of higher-order functions;

(iii) algebra-specific transformations that depend upon laws of a particular algebra;

(iv) architecture-specific transformations that depend upon representation equivalences or operations of the underlying implementation architecture.

Improvements can be gained in each of these categories, but they cannot be done all at once.

Generic transformations include deforestation (the elimination of intermediate data structures), and fusion of control structures common to two functions that are sequentially composed. For example, consider this expression that calculates the length of the catenation of two lists,

$$length(append(xs, ys))$$

In evaluating this expression, the list formed by appending the two sublists is needed only to provide a single list for analysis by the *length* function. If *length* were transformed to take account of the fact that the length of a catenation is the sum of lengths of its two components, then the catenation calculation could be avoided. In fact, this transformation does not require a special theorem about the functions *length* and *append*; it requires only their definitions, expressed as list reductions, or structural recursions over the list datatype. The above expression can then be transformed into an equivalent expression that accumulates the sum of the number of elements in the first list, starting not from zero, but from the length of the second list.

A remarkable fact is that the transformation itself is independent of the structure of the list datatype. It uses this structure only as data. The same generic transformation is also effective on an expression that calculates the number of nodes in a binary tree that is formed from a pair of trees by substituting the second tree for leaves of the first, or on a calculation of the number of identifiers in a phrase of a context-free-language formed by substituting one phrase in place of identifiers in another. In each case, the transformation produces a new counting function that avoids constructing the intermediate data structure.

Fusion and deforestation transformations are derived from the Promotion Theorem [10], which has instances in all structure algebras. The explicitly algebraic formulation of ADL programs makes it particularly easy to test for the presence of conditions under which these transformations apply. HOT (Higher-Order Transformations) is a new tool that works on ADL programs to carry out generic transformations automatically. Experience with HOT has been extremely favorable, although it can be frustrated by programs containing deeply nested conditional or case expressions, which are the bane of program transformation technology.

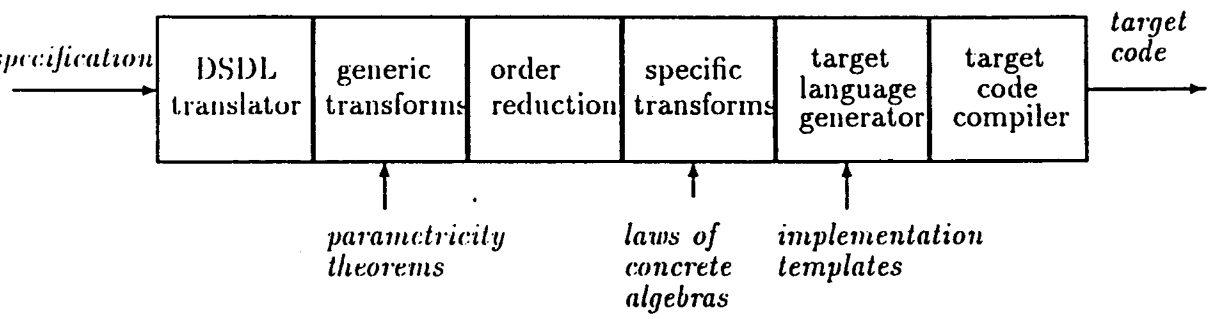


Figure 2—Translation and transformation pipeline

Order-reduction is the translation of higher-order functional programs to equivalent, first-order programs. A key step is the specialization of higher-order functions to the actual arguments that they take in the instances of their application in a program. For instance, the higher-order function *map* applies a function given as ban argument to every element of a list given as a second argument. If a program contained an application such as *map foo xs*, order reduction would replace this use of *map* with a new function in whose definition *foo* appears as a free identifier, rather than being passed as an explicit parameter. The original expression would be replaced by the new, first-order application, *map_foo xs*.

Order-reduction transformations have a long history, but there is a dearth of implemented tools. We have implemented an order reduction tool that we call PEP, which integrates a package of order-reduction algorithms that have been described in the literature.

Algebra-specific transformations make use of algebraic laws such as associativity and commutativity of specific operators. They are attempted after generic transformations have been performed. An example of an algebra-specific transformation is the recursion-to-iteration transformation of list reductions. For example, when the *reverse* function is defined as a list reduction, its recursive formulation is

$$\begin{array}{lll} \mathit{reverse}\ xs = \textbf{case}\ xs\ \text{of} & & \\ \quad \mathit{nil} & \Rightarrow & \mathit{nil} \\ \mid\ \mathit{cons}(x, xs') & \Rightarrow & \mathit{append}(\mathit{reverse}\ xs', \mathit{cons}(x, \mathit{nil})) \end{array}$$

However, given the following three laws about *append*, which might be found in a law library,

$$\mathit{append}(u, \mathit{nil}) = u \qquad (1)$$

$$append(nil,\ v) \;=\; v \qquad (2)$$
$$append(append(u,\ v),\ w) \;=\; append(u,\ append(v,\ w)) \qquad (3)$$

the recursion-to-iteration transformation for list reductions will find an equivalent formulation:

$$reverse\ xs = rev\ xs\ nil$$

where

$$\begin{array}{lll} rev\ xs = \textbf{fn}\ u \Rightarrow & \textbf{case}\ xs\ \textbf{of} & \\ & \quad nil & \Rightarrow\ u \\ & \mid\ cons(x, xs') & \Rightarrow\ rev\ xs'\ cons(x,\ u) \end{array}$$

Algebra-specific transformations, including the one illustrated above, are performed by ASTRE [3], a flexible transformation system based upon term-rewriting. Algebra-specific transformations are the most difficult of the four categories to automate, but they are of great importance for algorithm improvement.

The last category, that of architecture-specific transformations, is typically implemented in the code generation phase of an optimizing compiler. We have not found a need to implement because our system generates Ada as its target code, and the style of Ada that is generated is amenable to the code-improvement transformations built into many Ada compilers.

4 Specifying an implementation

First-order SML program modules are not difficult to compile to most target languages. The type system of SML is useful in guiding an implementation in terms of a strongly typed target language. The significant parameters of an implementation are:

- the targeted implementation language,
- the intended realizations of concrete algebras,
- interfaces with a host software architecture.

Realizations of concrete algebras in terms of the target language are specified in a set of *Implementation Templates* [14, 13]. Interfaces are prescribed in an environment specification. Both specifications are input to a tool called the Program Instantiator, which converts suitably restricted SML modules into packages of target language code. This step completes the generation of functioning software from a specification given in a DSDL.

5 Anticipated benefits of a generator technology

We have described a nascent technology for producing software component generators. Although it can now be demonstrated for a restricted domain of applications, continued research and development effort will be needed to realize its full potential. The major benefits that can be expected as the technology matures include:

- Significantly improved productivity of engineers during development and more particularly, throughout the maintenance phase of a software life cycle.
- Standardization of software component types, without requiring standardization of components themselves.
- An improved process for negotiating requirements with users and clients. Users can participate in the design of DSDL's for new domains.
- Improved ability to reason about critical properties of software components, including the use of mechanized verification.
- Reduced incidence of delivered defects.

6 Integrating software generation into a development process

The software generation technology we have described fits naturally into a domain-specific software architecture framework [1]. In such a framework, the components of a domain-specific architecture are intended either to be retrieved from an archive of reusable components or to be generated from specifications of their required function and of the environment in which they are to operate. In component reuse, an off-the-shelf module is installed without modification, but an interface must be constructed to match the component to a new environment. In component generation, the environment specification becomes a parameter to the generator and a new component is generated to fit. In a validation experiment, the SDRR method is being applied to construct a generator for message validation and translation components of the PRISM architecture for command and control systems.

Software evolution describes the entire process of change of a software module or system, including its requirements, its specification and design, and its implementation, throughout its lifetime [9]. Prototyping has been advocated as a technique for managing software evolution [8]. The SDRR method for designing and implementing software component generators is consistent with this view of evolution, affording early prototyping, design capture, and control of change.

The SDRR method is compatible with a measured development process. The tasks of designing a DSDL, defining its computational semantics, tailoring the transformation strategies to be used with a generator, and specifying its implementation are separable and separately measurable. Change is controllable and the effect of a change can be assessed accurately.

7 Next steps

The research we have described here holds the promise of a new technology that will support the widespread use of program generation to create components of software systems. To make this vision a reality, the prototype development system needs to be brought to further maturity, demonstrated on a wider range of applications, and transitioned into the software industry. We look forward to this challenge.

References

[1] The domain-specific software architecture program. Technical Report CMU/SEI-92-SR-9, Carnegie-Mellon Software Engineering Institute, June 1992.

[2] Jeffrey Bell et. al. Software Design for Reliability and Reuse: A proof-of-concept demonstration. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, March 1994. (To be presented at *Tri-Ada'94*.)

[3] Francoise Bellegarde. A transformation system combining partial evaluation with term rewriting. In *Higher Order Algebra, Logic and Term Rewriting (HOA '93)*, volume 816 of *Lecture Notes in Computer Science*, pages 40–58. Springer-Verlag, Sept. 1993.

[4] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO Series F*. Springer-Verlag, 1986.

[5] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Twente, The Netherlands, February 1992.

[6] James Hook and Tim Sheard. A semantics of compile-time reflection. Technical Report 93-019, Department of Computer Science and Engineering, Oregon Graduate Institute, November 1993.

[7] Richard B. Kieburtz and Jeffrey Lewis. Algebraic Design Language—Preliminary definition. Technical report, Pacific Software Research Center, Oregon Graduate Institute of Science & Technology, January 1994.

[8] Luqi. Software evolution through rapid prototyping. *Computer*, pages 13–25, May 1989.

[9] Luqi. A graph model for software evolution. *IEEE Trans. on Software Engineering*, 16(8):917–927, August 1990.

[10] Grant Malcolm. Homomorphisms and promotability. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 335–347. Springer-Verlag, June 1989.

[11] Lambert Meertens. Algorithmics—towards programming as a mathematical activity. In *Proc. of the CWI Symbposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[12] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, August 1991.

[13] Dennis Volpano. *Software Templates*. PhD thesis, Oregon Graduate Institute, October 1986.

[14] Dennis Volpano and Richard B. Kieburtz. Software templates. In *Proceedings Eighth International Conference on Software Engineering*, pages 55–60. IEEE Computer Society, August 1985.

STeP: The Stanford Temporal Prover

Zohar Manna
Computer Science Department, Stanford University

1 System Description

The Stanford Temporal Prover, STeP, supports the computer-aided formal verification of concurrent and reactive systems based on temporal specifications. Reactive systems are systems that maintain an ongoing interaction with their environment; specifications of reactive systems are typically expressed as constraints on their behavior over time. Unlike most systems for temporal verification, STeP is not restricted to finite-state systems, but combines model checking with deductive methods to allow the verification of a broad class of systems, including parameterized (N-component) circuit designs, parameterized (N-process) programs, and programs with infinite data domains. In short, STeP has been designed with the objective of combining the expressiveness of deductive methods with the simplicity of model checking.

Our development efforts have been focused on the following areas: First, in addition to the textual language of temporal logic, the system supports a structured visual language of *verification diagrams* [MP94] for guiding, organizing, and displaying proofs. Verification diagrams allow the user to construct proofs hierarchically, starting from a high-level, intuitive proof sketch and proceeding incrementally, as necessary, through layers of greater detail.

Second, the system implements powerful techniques for automatic *invariant generation*. Deductive verification almost always relies on finding, for a given program and specification, suitably strong (inductive) invariants and intermediate assertions. The user can typically provide an intuitive, high-level invariant, from which the system derives stronger, more detailed, *top-down invariants*. Simultaneously, *bottom-up invariants* are generated automatically by analyzing the program text. By combining these two methods, the system can often deduce sufficiently detailed invariants to carry through the entire verification process.

Finally, the system provides an integrated suite of simplifications and decision procedures for automatically checking the validity of a large class of first-order and temporal formulas. This degree of automated deduction is sufficient to handle most of the verification conditions that arise during the course of deductive verification—and the few conditions that are not solved automatically typically correspond to the critical steps of manually constructed proofs, where the user is most able to provide guidance.

An overview of the STeP system is shown in Figure 1. Inputs into STeP are a reactive system, represented by a program or a hardware description, and a property to be proven about the program, represented by a temporal logic formula. At present verification can be performed either by the model checker or by deductive means. In the latter case, the user typically provides a verification diagram to guide the proof. The system generates verification conditions from the diagram and will simplify most conditions, if not all, automatically. Additional user guidance may be provided by means of the interactive prover. For a more extensive description of the STeP system and examples of verified programs, the reader is referred to [MAB+94].

2 Application to Large-Scale Systems

If formal methods are to be successfully applied to large-scale systems, it is imperative that they be *compositional*, i.e., components of the system can be individually specified and the properties of the entire system can be deduced from these individual specifications. Recent advances in the compositional verification of reactive systems promise several advantages for the development of large-scale systems [Cha93]. First, verification of a large system can be reduced to the verification of the system's components, greatly simplifying the computational and conceptual difficulties associated with verifying large systems. Second, component libraries can be constructed and verified, providing a formal framework to support software reuse. Third, the process of compositional verification naturally requires an analysis of how each system component depends upon its environment, thereby capturing the information required for component-wise program evolution, in particular enabling maintenance of module and hence system integrity.

STeP already contains most of the building blocks necessary to perform compositional verification. We are currently extending STeP to support full-fledged compositional verification. We are also designing compositional methods for software synthesis and development.

Although full verification of large-scale systems requires powerful decomposition techniques, semi-formal analysis of large-scale systems is already feasible

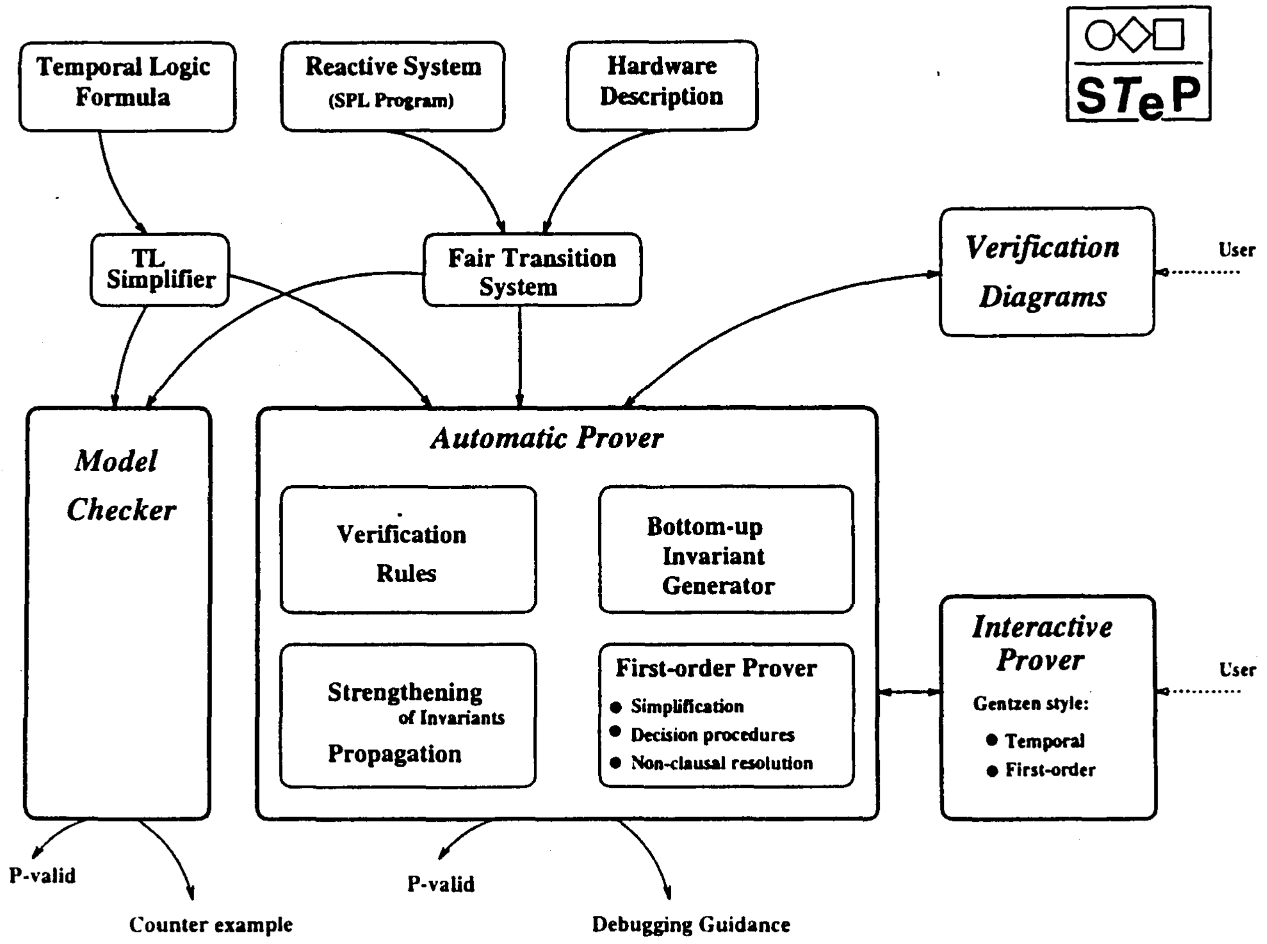


Figure 1: An overview of the STeP system

with the current system. Given a set of modular specifications, which together imply the overall system's requirements, each module can be verified separately. Each individual modular specification should explicitly state the assumptions about its environment and the commitments towards its environment. Explicitly stating these also enables automatic maintenance of local consistency of modules, as well as interface consistency between modules over the lifetime of the software. Other approaches towards handling large-scale systems that are feasible with current technology include abstraction (to verify existing systems) and refinement (to perform incremental design). Verification rules for refinement were recently published [KMP94] and are currently being implemented. Several other semi-formal consistency checking tools can be identified, e.g., automatic tracing of variables to identify the impact of modification of those variables, and similar syntactic checks. Application of these semi-formal analysis tools greatly increase the practicality of static analysis of large-scale systems.

Current research in our REACT group at Stanford is directed towards extending the STeP system in various other directions to obtain a general support tool for the design, analysis and verification of large-scale concurrent, real-time and hybrid systems. With respect to design we are working on algorithmic synthesis of reactive modules [AM94] and synthesis of concurrent programs. Another aspect of design and analysis we are studying is the application of automatic debugging techniques. In parallel we are working to extend the verification techniques to hybrid systems [KHMP94]. To aid in maintenance of large systems, we are exploring proof representations, in particular the creation of dependency structures, that enable one to keep track of which properties are dependent on which parts of the code. This line of research may also result in a contribution towards reusable components.

Much attention is also being paid to the user interface and the useability of the system. In addition to the verification diagrams mentioned before, a graphical user interface for specification of hybrid systems, tailored towards engineers, has been proposed [SM94]. To enhance readability and hence reviewability of proofs a LaTeX interface is provided to convert output to LaTeX automatically.

Summarizing, STeP, although still at an early stage of development, has the potential of becoming a powerful tool for the application of formal methods to programming-in-the-large, based on its emphasis on compositionality and its ability to support an adjustable degree of formality.

References

[AM94] A. Anuchitanukul and Z. Manna. Realizability and synthesis of reactive modules. In D.L. Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 156–168. Springer-Verlag, 1994.

[Cha93] E. Chang. *Compositional Verification of Reactive and Real-Time Systems.* PhD thesis, Department of Computer Science, Stanford University, Stanford, California, 1993.

[KHMP94] A. Kapur, T.A. Henzinger, Z. Manna, and A. Pnueli. Proving properties of hybrid systems. In *Proc. Third Int. Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, Lübeck, Germany, September 1994. Springer-Verlag.

[KMP94] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 273–346. Springer-Verlag, 1994.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP: the Stanford Temporal Prover. Technical report, Computer Science Department, Stanford University, August 1994.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In *Proc. of the* 11^{th} *Annual Symp. on Theoretical Aspects of Computer Science*, volume 789 of *Lecture Notes in Computer Science*, pages 726–765. Springer-Verlag, 1994.

[SM94] H.B. Sipma and Z. Manna. Specification and verification of controlled systems. In *Proc. Third Int. Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, Lübeck, Germany, September 1994.

Representing Rationale in Formal Systems Development

Balasubramaniam Ramesh
Code SM/RA
Naval Postgraduate School
Monterey, CA 93943
e-mail: ramesh@nps.navy.mil

Abstract

A major concern with the use of formal specification based software development is the difficulty users face in arriving at specifications. Further, with this paradigm, the process of maintenance is done at the level of specifications. Therefore, it is essential to capture the essence of the process of arriving at formal specifications, so that they can be easily understood and maintained. We have implemented the REMAP model for representing rationale in Concept Demo, a formal software development environment, providing a graphical browser for instantiation, browsing and modification of process knowledge components and mechanisms to reason with this knowledge. Our approach aims at providing comprehensive traceability between informal and formal aspects of systems development.

1 Introduction

Based on a recent survey on the use of formal methods in industrial use Gerhart et al [1] suggest that integration of formal methods with existing industry practices is extremely important for the long-term success of formal methods. Such an integration facilitates smooth transition from current practices to the use of formal methods. Besides, in many critical applications, where formal methods have high chance of successful application, the understandability of the method and the specifications is extremely important to the end users for its successful adoption.

An important component of such an integration is the maintenance of *process knowledge* or the reasons behind the creation of artifacts, i.e., *design rationale* (DR)[1]. The capture and use of design rationale is widely recognized to be essential for the design and maintenance of large systems. Effective mechanisms for dealing with design rationale must be able to capture and reason with formal as well as informal knowledge.

[1]Throughout this paper the term *design* is used to refer to any activity that leads to the creation of artifacts. Potts and Burns [2] note that even the early phases of the systems development life cycle involve creation of intermediate artifacts, and therefore, the term *design* is used to denote these activities as well.

2 The Representation of Rationale

DR can be represented in a variety of ways, from "mathematically" formal representations (e.g., transformations that can formally derive one problem state from another) to very informal representations (e.g., design notebooks that record rationale in natural language). Formal representations of DR are feasible in domains that have formal domain models, and in systems design tasks where the semantics are well defined. The availability of formal representations facilitates automated reasoning with DR knowledge. In the design of large scale systems - where the size and complexity of DR knowledge grows exponentially - the facility for automated inference and support can be extremely valuable.

However, attempts at formal representation of DR are constrained by several limitations. The domain knowledge needed to capture formal representations may not be readily available. In such situations, the knowledge must be acquired. The acquisition of DR knowledge is often made through informal means, e.g. videotapes of meetings. This knowledge is then converted to formal representations of process knowledge. This conversion process is subject to two sets of problems. The process is extremely labor-intensive, and therefore may be infeasible to implement in many design situations. Second, the act of representation entails making judgments about the level of granularity in which the information should be represented. Overly large grained representations may result in loss of useful detail, while overly fine-grained representations may create "trivial" knowledge wherein the benefits obtained from the finer grain do not warrant the cost of creating such knowledge. Existing literature does not offer demonstrably effective decision rules for making judgments about granularity.

More fundamentally, the nature of the informal information often means that significant amounts of DR knowledge do not lend themselves to formal representation. Since design is primarily a collaborative process [3], informal DR knowledge often consists of deliberations among individuals engaged in the process. When individuals interact, they communicate through multiple channels. While some channels are explicit (e.g. talk), others are used in implicit ways. For example, the social significance and information content of gestures such as nods and looks have long been recognized to be an integral part of human communication [4]. Communication is also effected through passive forms, such as awareness [5]. Furthermore, design deliberation sessions often serve as a forum for the resolution of social and motivational issues such as conflicts among stakeholders. Such issues are often handled by groups through tacit or unconscious mechanisms that resist the explicit treatment in knowledge bases [3]. As Anderson et al [6] point out, attempts to represent informal knowledge through formal tools and notations can result in thin descriptions [7], with the consequence that much of the meaning embedded in such information is then lost.

Informal representations of DR can alleviate many of these problems. Since much DR knowledge is captured through informal means, mechanisms for informal representations make the task of creating DR much easier. Second, informal representations enable the retention of information in its most complete form, thereby facilitating the creation of thick descriptions [8]. Recording human interaction in such forms allows access to the richness and complexity of social action, thereby allowing particular events to be scrutinized repeatedly and subjected to detailed inspection (Heath and Luff 1992b). Thick descriptions enable the user of DRs to grasp the subtleties, tacit and mutual knowledge, and glean descriptions of work practices that are otherwise not made explicit [9]. Finally, while formal representations can only be used by individuals who are familiar with the rigors of such formalisms, informal representations can be used by a much wider set of use! rs.

However, the classification, indexing, retrieval and use of informal representations is problematic. Given the volume of knowledge generated in large projects, the lack of appropriate navigation devices can constitute a significant impediment to the use of DR. Moreover, though understandable by humans, such information representations are not amenable to "computation", i.e., unlike formal representations, they cannot be used by the computer to provide automated inference and support. Thus, while informal representations hold much promise with respect to the information they contain, the difficulty of accessing and reasoning with such information can undermine its utility. In summary, then, formal and informal representations of DR complement each other in their respective strengths and weaknesses. Informal representations are easy to capture, whereas formal representations, can be manipulated by well-defined inference procedures. Thus, effective schemes for the capture and use of DR should seek to combine the advantages of both forms of representations.

3 Our Approach

The process of defining formal specifications from the initial set of informal requirements can be thought of as a deliberation. We have developed the REpresentation and MAintenance of Process knowledge (REMAP) model for representing such deliberations. The REMAP model was developed to support the capture and reuse of design rationale knowledge during systems development. The model represents deliberations that could occur in any systems development activity. Our work extends the Issue Based Information Systems (IBIS) framework for representing deliberations. REMAP model and mechanisms are described in detail elsewhere [10]. The process knowledge or rationale behind specifications can be invaluable in the context of changing requirements and assumptions. REMAP provides a special purpose reason maintenance system to manage the dependencies among design rationale and formal specification components.

We have implemented REMAP in Concept Demo, a prototype environment developed as a part of the the Knowledge Based Software Development (KBSA) system. The KBSA under development at the USAF Rome Laboratory is intended to be a formally-structured knowledge based, software design, development and maintenance tool that encompasses the entire software development life cycle. This paradigm combines formalization with automation to achieve four distinguishing features:

- incremental, formal and executable specifications

- formal implementation where verification and validation arise from the implementation development process
- enforced project management policy maintaining consistent relationships between various software objects
- high level system development and maintenance accomplished at the requirements and specification levels

The objective of our effort is to provide an environment for the capture and reuse of process knowledge as formal specifications get defined and modified over the life cycle, providing comprehensive traceability between informal and formal aspects of systems development. In the context of formal systems development using transformations, traceability at the design and implementations stages could be a by-product of the process. However, in addition to maintaining traceability of specifications to design artifacts, it is important to capture information about where requirements came from. Current practices in requirements traceability address traceability essentially only after the requirements specification phase of systems development. Recent research suggests that a primary reason for problems with current traceability efforts is the lack of pre-requirements specification traceability [11]. REMAP supports pre-requirements traceability by providing linkages between formal specifications and the informal requirements, needs or objectives that motivate them. Such information is important in understanding, communicating and modifying specifications throughout the systems development life cycle.

4 Support for Systems Development

Our research identified several task specific types of support that can be provided to various stakeholders involved in systems design. These include domain knowledge reuse, design replay, project management, evolutionary systems design with changing requirements and assumptions. We have developed a prototype of an environment to support these activities. Reasoning mechanisms used in our research include reason maintenance and temporal reasoning to provide support for various stakeholders.

4.1 Discussion

The empirical evaluation of the usefulness of the REMAP model and mechanisms are of extreme importance. As an important step, the REMAP model and mechanisms are being incorporated in several candidate software development environments such as Concept Demo. This project will enable the evaluation of REMAP in the context of software development based on formal specifications. Experimental evaluations of the benefits of capturing rationale in the context of formal software development are the focus of ongoing work. Our evaluation in this context would focus on the feasibility of capturing and maintaining design rationale information as a part of a traceability scheme in large scale real-time systems development efforts.

References

[1] S. Gerhart, D. Craigen, and T. Ralston, "Observations on industrial practice using formal methods," in *proceedings of the 15th international conference on softwrae engineering*, May 1993. Baltimore, MD.

[2] C. Potts and G. Bruns, "Recording reasons for design decisions," in *Proceedings of the 10th International Conference on Software Engineering*, (Singapore), pp. 418–426, April 1988.

[3] G. Fisher, J. Grudin, A. Lemke, R. McCall, J. Ostwald, B. Reeves, and F. Shipman, "Supporting indirect collaborative design with integrated knowledge-based design environments," *Human-Computer Interaction*, vol. 7, pp. 281–314, 1992.

[4] C. Heath and P. Luff, *Explicating Face-to-face interaction*, ch. 2. G. Gilbert (ed), Researching Social Life, London: Sage, 1992.

[5] P. Dourish and S. Bly, "Portholes: Supporting awareness in a distributed work group," in *Proceedings of the ACM Conference on Human Factors in Computer Systems CHI '92,*, (Monterey, CA), 1992.

[6] R. Anderson, C. Heath, P. Luff, and T. Moran, "The social and the cognitive in human-computer interaction," Technical Report EPC-91-126, Rank Xerox EuroPARC, Cambridge, U.K., 1991.

[7] Ryle, *Concept of Mind.* Harmonds Worth, 1949.

[8] C. Geertz, *Thick Description: Toward an Interpretive Theory of Culture*, ch. 3. Interpretation of Cultures, New York: Basic Books, 1973.

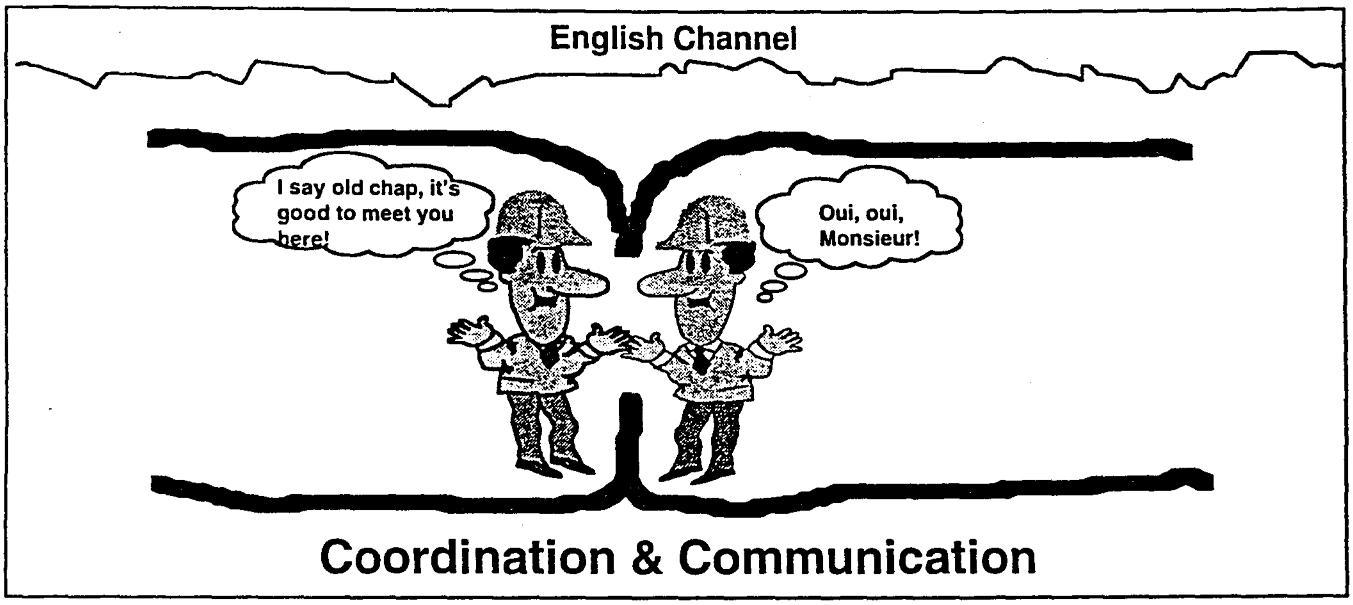
[9] B. Jordon, "Technology and social interaction: Notes on the achievement of authoritative knowledge in complex settings," Technical Report IRL92-0027, Institute for Research on Learning, Palo Alto, CA, 1992.

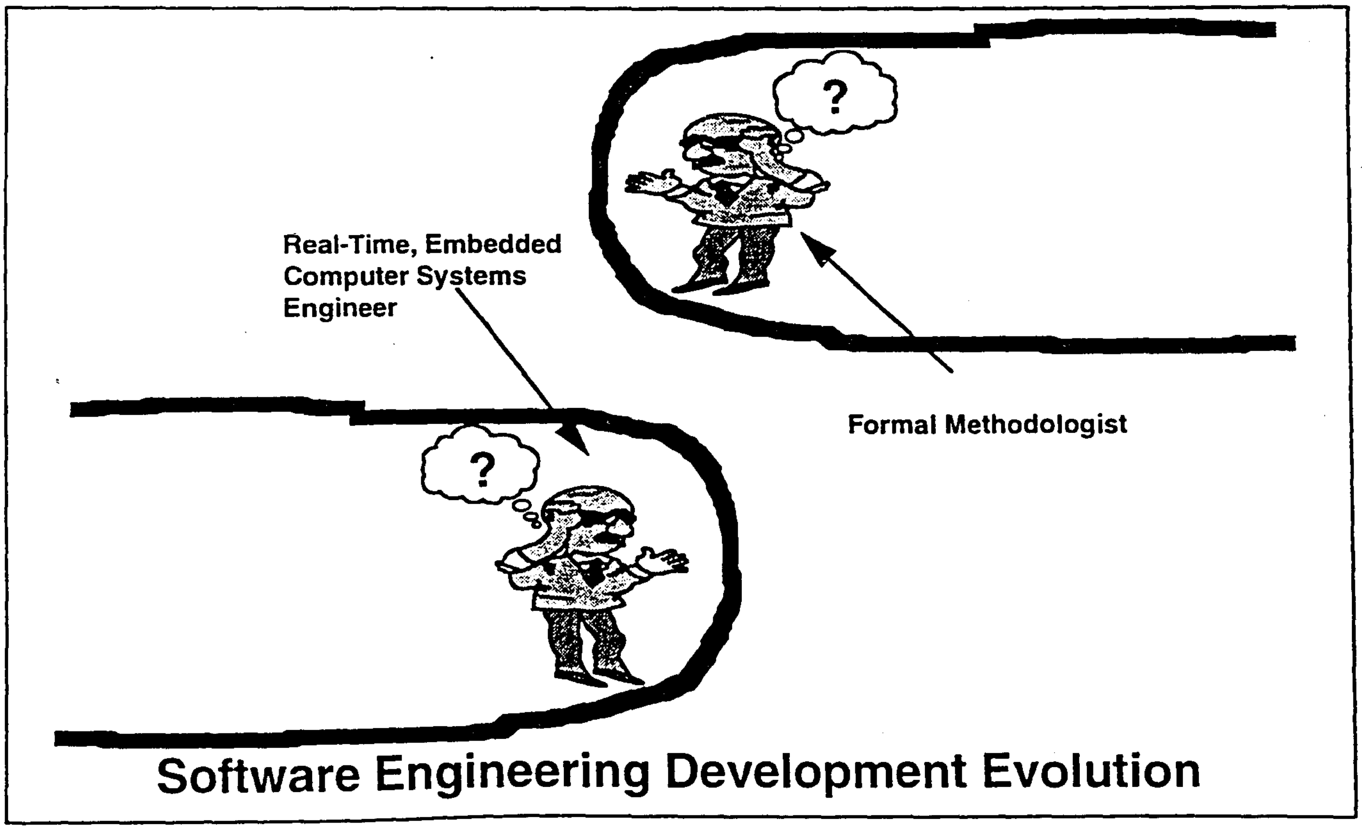
[10] B. Ramesh and V. Dhar, "Supporting systems development using knowledge captured during requirements engineering," *IEEE Transactions on Software Engineering*, June 1992.

[11] O. Gotel and A. Finkelstein, "An analysis of the requierments traceability problem," technical report, Imperial COllege, London, UK., 1993.

Reengineering Real-Time Embedded Computer Software With the McCabe Tools

B.H. Ace Roberts
Software Engineering Directorate
U.S. Army Missile Command
Redstone Arsenal, Alabama


English Channel
I say old chap, it's good to meet you here!
Oui, oui, Monsieur!
Coordination & Communication


Real-Time, Embedded Computer Systems Engineer
Formal Methodologist
?
?
Software Engineering Development Evolution

REENGINEERING REAL-TIME EMBEDDED COMPUTER SOFTWARE WITH THE McCABE TOOLS

In the 1994 Monterey Workshop announcement it is stated: "DOD and the computer industry urgently need software systems that meet user needs effectively and reliably.", and "The purpose is to assess current efforts, to identify results and directions for increasing the degree of automation, to build a common understanding about the integration of methods and tools, and ultimately to help bring formal methods into practical use." The following is a case history of the practical problems encountered by the U.S. Army Missile Command (USAMICOM) Software Engineering Directorate (SED), the methodology employed, and the automated tool procured to assist in the solution of these problems.

In 1992, USAMICOM awarded a contract for the Ground Base Sensor (GBS) system which utilized Non-Development Item (NDI) Hardware and Software. As a rule, software for an NDI System proposed by a contractor is either only partially developed (i.e., the old system consists of incomplete and inadequate requirements with regard to the new application, and thereby requires: system engineering and/or re-engineering; a new design or redesign of the software; new code, code deletion or modification; additional testing, e.g., regression testing); and/or is lacking a full set of documentation; deficiencies, all found in the GBS NDI software. Also, the computer language was ULTRA 16, an Assembly Language (circa 1972). The GBS Project Office wanted assurance regarding the software quality (e.g., reliability, maintainability, supportability, and suitability). The USAMICOM SED recommended that a code analysis tool be obtained to analyze, evaluate, and determine the existing NDI software's relative quality and to monitor any subsequent additions, deletions, and/or modifications to the software. Most of the current CASE Tools and code analysis tools are designed and built for MIS/commercial systems, not real-time, embedded computer systems. After examining the ULTRA 16 source code and the ULTRA 16 Programmer's Manual, McCabe & Associates said they could "build an ULTRA 16 parser as a front end" for the McCabe Tools. Because of the real-time, embedded computer nature of assembly language, when SED obtained the services of McCabe & Associates to develop an ULTRA 16 parser, some of the features required to address and account for the problems associated with real-time, embedded computer systems were designed into the ULTRA 16 parser.
In Oct 1992, the USAMICOM SED began analyzing the contractor's existing NDI Source Code with the McCabe Tools and Ultra 16 parser.

There are four main differences between embedded, real-time systems and conventional/idealized computer systems:
1) the executive/operating system.
2) the logical flow between the executive and a module and/or between the modules.
3) Timing constraints; including interrupts and polling.
4) Real-world/real-time environment interaction (input/output); A/D (Analog to Digital) and D/A (Digital to Analog).

The differences in the executive system for an idealized/ conventional vs. a real-time, embedded system are:
A) An idealized/conventional computer system is event driven and any timing requirements can be simulated.
B) A real-time, embedded computer system is both time and event driven; i.e., it operates within time frame constraints, asynchronous and synchronous, and has various timers and interrupts and/or polling (interrogating the status of a device periodically).

Sensors sample/poll the system environment and provide inputs to the embedded computer system. Outputs from the embedded computer system to servos or other electro-mechanical systems provide control for the system. The response of the embedded computer system to the environmental inputs is both magnitude and time critical; an interrupt must be serviced and the response/output issued within a specified time period, plus the response/output must be both accurate and dynamically stable. The real-time nature of the system is manifested in the hierarchial structure of the **Battlemap**. The analysis of a conventional/idealized computer system vs. a real-time, embedded computer system shows that the main difference, as represented by the McCabe tools output, is at the **Battlemap** level. There are many modules shown in the first level of the **Battlemap** which do not call other modules or are not called by other modules. The USAMICOM SED has worked with McCabe & Associates personnel to explain these anomalies and how to portray them properly within a real-time system context. Analysis of our real-time, embedded computer system **Battlemap** revealed that the many modules at the top level are due to the following reasons:
1) "hooks" (calls inserted in the source code for tasks that may not be defined or designed at the current time; i.e., the inserted call "holds a place" for a subroutine to perform a known necessary future task, but not written yet);
2) utilities (generic code and/or code already written but not yet determined where it will be used/called);
3) interrupts;
4) counters;
5) indirect addressing/stack file techniques (intricate code dynamically determining logical flow between modules);
6) "orange software" (for test purposes; i.e., stubs and drivers);
7) redundant code (as a result of how McCabe & Associates treat labels in the ULTRA 16 assembly language source code);
8) data bases (not executable source code); and
9) "jump" commands which dictate logical program flow.

Several features of the McCabe Tools were utilized to model the unique characteristics of the real-time, embedded system: "Hooks", utilities, data bases, "orange software" modules, and modules resulting from redundant code can be deleted from the **Battlemap** with the OMIT toggle/command in the EXCLUDE feature. Other modules resulting from convoluted logical control flow techniques; e.g., indirect addressing/stack files, can be handled by the ATTACHTO toggle/command in the McCabe Tools EXCLUDE feature. Since the development of the McCabe Tool ULTRA 16 Assembly Language parser, several additions/ modifications have been requested by USAMICOM

SED. A new feature was developed to represent the logical program flow as dictated by the ULTRA 16 Assembly Language "jump" (GO TO) command. Real-time systems use the "jump" command instead of "calls" (in idealized /conventional systems) because of timing constraints; i.e., a conventional/idealized system's top level has only one module, the executive/operating system controlling all other modules by "calls" to the modules and subsequent "returns" from the modules to the executive/operating system. The time required to execute a "jump" is significantly less than the time for a "call" and "return". "Jumps", being time efficient, will perform the program iteration within the required amount of time during which: normal computations are done, interrupts and pollings are serviced and processed, the respective commands and graduated signals are issued, and the various timing constraints met. The result is an executive/operating system made up of a series of modules strung together by "jump" commands rather than a single executive/ operating system (in a conventional/idealized computer system). McCabe & Associates added a "-J2C" option which treats a "jump" instruction as a module "call", if the location specified by the "jump" is outside the file. Since a file may produce many modules, "jumps" from a module to other modules within the file are not recognized and implemented. Showing the control flow between these modules requires the usage of the ATTACHTO option in the EXCLUDE feature, OPTIONS menu. Close coordination between the USAMICOM SED and McCabe & Associates personnel produced a requested lines of code counter which identified and counted the Source Lines Of Code (SLOCs), blank lines, comment lines, and data base lines.

The USAMICOM SED has used the standard features of the McCabe Tools in developing a "quasi-formal" software development methodology:
1) graphical representations: hierarchial chart describing overall architecture (**Battlemap**) and "red" modules flow diagrams;
2) "red" modules source code listings and test paths lists/graphs;
3) module metrics lists including: McCabe Cyclomatic Complexity, Essential Complexity, Module Design Complexity, etc.;
4) alphanumerical modules lists;
5) numerical modules lists;
6) context lists (showing files from which modules emanated to monitor the development of the system software and to analyze and evaluate the quality of the resulting product); and
7) SLOCs for each file;

The metrics in the alphanumerical module lists and numerical modules lists and the SLOCs were used to identify modules with high complexities and/or large modules. These modules were analyzed with the metrics, **Battlemap**, module flow diagrams, SLOCs, and source code listings to determine their maintainability, supportability, and reliability. A "red" module (McCabe Cyclomatic Complexity greater than 10, and an Essential Complexity greater than 8) is then examined to determine how (and whether) the module should be reengineered; i.e., broken up and restructured and/or subdivided. The context lists are used to determine from which file each module is produced. The SLOC counter was used to measure the percent change between the first and the most recent build and to determine if the DOD/AMC standards/ regulations for the percentage of software modification and addition have been exceeded,

Recent efforts have consisted of using the **Codebreaker** tool to identify redundant code. During a McCabe & Associates' Automated Reverse Engineering seminar, an attempt was made to produce a Software Design Document (SDD) in accordance with DOD-STD-2167A and DI-MCCR-80012A [the SDD DID] using the McCabe Tools and either FRAMEMAKER or INTERLEAF. This exercise was a limited success; the technology hasn't been developed and/or integrated to a point where an SDD can be produced with a minimal amount of effort. Therefore, the USAMICOM SED has generated a Small Business Innovative Research Topic Submittal for the development of "An Automatic Document Generator" using either the source code and/or a Programming Design Language (PDL) code, plus additional inputs for the interface documents. An intended future use of the McCabe Tools will be to convert functional decomposition ULTRA 16 assembly language source code to object based Ada or Object Oriented Ada 9x source code.

Using an automated tool, Independent Validation and Verification (I V & V) is more efficient, accurate, and comprehensive. Also a significant cost and labor savings was realized by using the McCabe Tools versus performing the I V & V tasks by the old manual labor methods. The COnstructive COst MOdel (COCOMO) in <u>SOFTWARE ENGINEERING ECONOMICS</u>, by Dr. Barry W. Boehm, was used to estimate the manpower for the SED I V & V activities. Historical data has established that the I V & V manpower is 20-30% of the Total Manpower for a software development project, as calculated by COCOMO. An average estimate of 25% of the Total Manpower and the labor rate of $ 110 K/year = $ 9.166 K/month was used. The overall net savings to the Government by the acquisition of the McCabe Tools was predicted to be <u>$</u> <u>1.56</u> <u>M</u>, based on the following calculations:

1. Operational Software: From the GBS proposal: 30.7K SLOCs

From COCOMO: $\text{MM (nom)} = 2.8(30.7)^{1.2} = 170.5$ Man-months

$\text{MM (adj)} = (3.105)(170.5) = 529.4$ Man-months

$\text{MM (analysis)} = (.25)(529.4) = 132.35$ Man-months

Cost [analysis] = (132.35)($ 9.166...K) = $ 1213K = $ 1.213M

2. Support Software: From the GBS proposal: 35.4K SLOCs

From COCOMO: $\text{MM (nom)} = 3.0(35.4)^{1.12} = 162.93$ Man-months

$\text{MM (adj)} = (1.67)(162.93) = 272.1$ Man-months

$\text{MM (analysis)} = (.25)(272.1) = 68.02$ Man-months

Cost [analysis] = (68.02)($ 9.166...K) = $ 624K = $ .624M
<u>Total Estimated Cost to do Analysis [manually]:</u>

```
                    Operational Software:  $ 1.213M
                        Support Software:  $  .624M
                                           --------
                                           $ 1.837M
```

3. An estimate was made: it would take 4 men working for six months to complete the I V & V on the software with the McCabe Tools, (4 men x 6 months = 24 Man-months [MM]).

```
Cost [analysis with: (24 MM)($ 9.166...K) = $ 220K = $ .22M
      McCabe Tools]
```

<u>Total Estimated Cost to do Analysis [with McCabe Tools]:</u>

```
                            Labor Cost: $ .220M
                     McCabe Tools Cost: $ .061M
                                        -------
                                        $ .281M
```

<u>Total estimated Savings using McCabe Tools:</u>

```
                  Analysis [manually]:   $ 1.837M
        Analysis [with McCabe Tools]: - $  .281M
                                         -------
                                         $ 1.556M ~= $ 1.56M
```

The above calculations' validity was proven when only two men were available to perform the I V & V tasks and the time to complete the I V & V tasks was approximately one year (12 months). For two men working 12 months: (2 men X 12 months = 24 man-months); a figure identical to the estimate of 24 man-months [MM] obtained in # 3. above, thus verifying the saving of $ 1.56M, as predicted.

Since the success of using the McCabe Tools on this program, USAMICOM SED has purchased Ada, C, C++, PLM, and FORTRAN parsers, and the McCabe Tools have been used by USAMICOM SED personnel to produce similar success on several other programs. The McCabe Tools could be used with rapid prototyping to examine and evaluate each prototype plus obtaining the metrics for each prototype. Thus, the McCabe Tools could be used to maximize a prototype's qualities and features while minimizing the weaknesses and deficiencies.

While it is necessary to combine formal methods and automated tools, it won't be sufficient for real-time, embedded computer systems until the marriage of formal methods and automated tools is consummated in a real-time, embedded computer system environment. Therefore, we must be pragmatic and concerned about real-time, embedded computer system environment constraints and features, and how they are represented, treated and modeled with formal methods and automated tools.

Temporal State Machines and Assertions: A Practical Framework for Handling Changes in Real-Time Systems*

Alan C. Shaw
Department of Computer Science & Engineering FR-35
University of Washington
Seattle, WA 98195

shaw@cs.washington.edu

August, 1994

1. Goals and Approach

Change is life. In the computer world, software requirements, designs, and implementations are evolving continuously in response to changes in technology, economics, competition, user needs, and other factors. Our aim is to develop and apply formal methods in the real-time software area, that can be used practically for specification and documentation, for fast prototyping and simulation, for monitoring and testing, and for verification and analysis, throughout the software life cycle. We argue here for accomplishing some of the above with particular versions of timed state machines and assertions, and present some evidence for the success of our methods.

State machine formalisms have a rich history of practical use in computer systems. They are particularly appropriate for specifying system behaviors because they have natural pictorial representations that appeal to working engineers, they are executable, they are amenable to analysis, they lead to implementations, and they have proven to be scalable. Assertional logics are the most popular class of techniques for describing system properties, probably because they are based on familiar and standard mathematical notions, they provide a natural language within which one can reason and, in principle, prove things about software, and they fit well with state machines, for example, for testing and verification.

Our approach employs communicating real-time state machines (CRSMs), behaviors represented as timed traces (histories) of input-output (IO) events, and assertions over these traces. We first briefly review these formal notations, and then discuss some practical applications and potential in software evolution, in particular, related to fast prototyping, monitoring simulations, verification, and implementation testing. More details can be found in the references [Shaw 92, 93, 94; Raju & Shaw 94; Raju 94]. Related works include [Harel 87; Jahanian & Mok 86, 89; Leveson et al. 92; Kramer et al. 93; Gabrielian & Franklin 91; Ostroff & Wonham 87; Lynch & Tuttle 88; Hoare 85]. Our work differs from these in the novel and complete way in which we have embedded timing behaviors into state machines, in our particular syntheses of machines and communications, and in our proposal for systematically using assertions from requirements through implementations.

* This work was supported in part by the National Science Foundation under grant number CCR-9200858 and by the Army Research Office under grant number DAAH04-94-G-0226.

2. CRSMs, Traces, Assertions, and Change

CRSMs are timed and universal state machines that communicate synchronously over unidirectional channels. Transitions between states are guarded commands; the execution time of a transition is described by a time interval. Every CRSM has a partner clock machine, also a CRSM, that provides a timeout mechanism and can be queried for the value of real-time. IO channels are the interfaces between CRSMs and subsystems of CRSMs. A collection of CRSMs is meant to specify a closed system consisting of both the computer system and its environment.

More formally, a CRSM is a tuple $M = (S, I, O, V, G, C, E, T, s_0)$. S is a finite set of states. I and O are each finite sets of input and output channels, respectively. V,G,C, and E are finite sets of variables, guards, commands, and expressions, respectively. C may contain IO commands or internal commands (programs or identifiers of physical activities); IO commands are of the form $ch(v)?$ or $ch(expr)!$ where ch is a channel or event class, v a variable, $expr$ an expression, "?" denotes a synchronous receive and "!" denotes a synchronous send (much like Hoare's CSP). T is a finite set of transitions $T \subseteq S \times S \times G \times C \times E \times E$; the two expressions (E) represent time bounds for the execution of the transition. s_0 is the start or initial state of M.

The behavior of a system is defined by a set of traces or histories over the IO channels. A trace $tr(ch)$ over a channel ch is a (possibly infinite) sequence of timed IO events:

$tr(ch) = < x_0 , x_1 , ... , x_i , ... >$

where the ith event on channel ch, $x_i = (v_i , t_i)$, is given by the value of the message v_i and the time of the communication t_i.

Safety and timing properties are expressed as assertions over these traces of communication events, using a notation based on Jahanian & Mok's real-time logic (RTL). RTL defines a function time(E,i) which returns the time of the ith event of class E, if it has occurred. For $tr(ch)$ above, we have t_i = time(ch,i). The RTL expression:

$\forall i : i \geq 1 \Rightarrow (\text{time}(ch,i) - \text{time}(ch,i-1)) = 10$

states that exactly 10 time units separate events of type ch.

Given a description of requirements, designs, and desirable properties in these notations, what is the range of changes that could occur? At the CRSM level, one could change time, commands, control logic, interfaces (channels), and create and delete machines. Changes in assertions might involve changes in any of the components of a trace. The problems are how to make changes easily, to propagate changes through the software cycle, and to examine the effects of changes.

3. Applying The Formalisms

3.1 Executing CRSMs

Simulators for CRSMs can be constructed in a fairly straightforward way to permit fast prototyping of requirements and designs. In particular, the effects of changes can be determined quickly, for example, allowing the designer or user to pose and answer a wide range of "what if" questions. Our implementation has a graphical editor for creating CRSMs; a C/C++ syntax is used to express state transitions. A system of CRSMs is translated into an interactive C++ program that simulates the machine, and allows user control and examination of the execution.

By "programming" the machines appropriately, it is easy to modify many parts of requirements and designs interactively during a simulation. Specifically, the values of variables can be set, for example those that define the state of the environment. An example might be a variable giving the rate of arrival of cars at an intersection in a traffic light controller system. Essentially, all of the example specifications given in our referenced papers have been validated with the simulator.

3.2 Monitoring of Event-Based Assertions

We believe that it is unrealistic to hope that program *proving* technology, either automatic or manual, will ever eliminate the need for debugging, especially for larger practical systems. However, the same logical formalisms can be used. In particular, a useful, and relatively novel, way to test or debug a specification or an implementation is through assertion checking at key points in their execution. Again, a major application of this feature is to debug and examine the effects of proposed or required changes.

In conjunction with the CRSM simulator described above, we have developed an event-triggered assertion monitor. An event-triggered assertion over a trace has the general form:

```
        when <event_expression> { Compute_and_Check_Assertion }
```

where <event_expression> names an IO event with an optional time offset, and "Compute_and_Check_Assertion" is a C/C++ program. There are functions to retrieve the elements of IO traces on every channel and a special Boolean *assert* function. During simulation, whenever an event_expression is satisfied, the corresponding code is executed.

For example, to check that a machine sends a "tick" message (the *null* message over the channel named *tick*) every 10 seconds, the constraint would be expressed*:

```
        when  tick
              {  t1 = time(tick, -1);  t2 = time(tick, -2);
                 assert( t1 - t2 == 10 ); }
```

where time(tick, -1) gives the time of the most recent *tick* event, and time(tick, -2) returns the time of the second most recent *tick* event.

All of our simulated examples have also been checked with appropriate assertions. Surprisingly (to us), as a consequence of a failed assertion, we found, and subsequently corrected, an error in a traffic light controller example. As an interesting and practical side benefit, the same event-triggered mechanisms can be used for performing other types of analyses of a design or of a prototype. For example, we have used these software tools to measure resource usage (e.g., a running average of the time between various events or the traffic on a channel).

3.3 Verifying Assertions

While we have explicitly de-emphasized verification as one of the major practical applications of formal methods, it is worth noting that some properties can be automatically verified provided the CRSMs are suitably restricted (e.g., variables range over finite sets of values). Such a verifier has been constructed and used to prove (through reachability analysis) many standard behavioral properties in our examples, such as delays, deadlines, mutual exclusion, and absence from deadlock.

* Some liberties have been taken with the syntax, for simplicity.

(Many safety properties, such as the last two, should be invariant to changes.) The usual state explosion problem is exacerbated with the addition of time, but can be controlled to some extent by some clever tricks with time intervals.

The verification work also produced one unforeseen application, and provides a good illustration of how this technology can be used for proving the correctness of small but non-trivial concurrent algorithms. Travis Craig has used our tools to specify and verify the correctness of a new real-time, queuing spinning lock algorithm for accessing storage in a multiprocessor system.

3.4 Monitoring of Implementations

Assertion monitoring can also be used at the implementation level. It also provides a simple basis for checking the consistency between specifications and implementations -- the *same* assertions can be employed for both. (Of course, the same interfaces must be used in each phase.) For the final system, assertions can be compiled into a run-time monitor that does the checking. The consistency application was developed after some work in applying RTL to run-time monitoring of distributed systems [Raju et al. 92]. It should be especially useful during the evolution of implementations and their specifications.

4. Concluding Remarks

Do the ideas and experiments scale up to larger systems? We have started to address this question in our recent work on specifications-in-the-large, where mechanisms for composing CRSMs into subsystems and larger systems are introduced. Similar state-based formalisms, but without time, have been used successfully for specifying larger systems, for example, for the aircraft collision avoidance system TCAS II [Leveson et al. 92]. Questions and techniques related to the scalability of assertions, i.e., composing and structuring them, need to be addressed in more detail.

A second question of special relevance to the theme of this paper is how to deal with changes more systematically. For some types of changes, a more formal incremental, analysis may be feasible, perhaps a perturbation analysis that indicates how robust a system is to small changes. For example, the frequency and size of messages (IO events) would be logical candidates for this approach.

We have presented some results and ideas for handling some aspects of the real-time software evolution problem through formal methods. The principal practical applications are dealing with change through fast prototyping of timed state machines, and using assertions over timed IO traces for monitoring requirements and designs to assure consistency and to validate both safety and timing properties.

Acknowledgment: Much of the recent work reported here was done jointly with Sitaram Raju. The research on the general use of assertions and on verification is part of Raju's Ph.D. dissertation [Raju 94].

5. References

[Gabrielian & Franklin 91] A. Gabrielian and M. Franklin, "Multilevel Specification of Real-Time Systems," *Comm. of ACM 34*, 5 (May 1991), pp. 50-60.

[Harel 87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming 8*, (1987), pp. 231-274.

[Hoare 85] C. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[Jahanian & Mok 86] F. Jahanian and A. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Trans. on Software Eng.*, Vol. SE-12, (September 1986), pp. 890-904.

[Jahanian & Mok 89] F. Jahanian and A. Mok, "Modechart: A Specification Language For Real Time Systems," IBM Tech Report RC 15140, November 1989.

[Kramer et al. 93] B. Kramer, Luqi, and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language," *IEEE Trans. on Software Eng.*, Vol. 19, No. 5, (May 1993), pp. 453-477.

[Leveson et al. 92] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese, "Requirements Specification For Process-Control Systems," TR 92-106, Computer Science Dept., UC Irvine, 1992.

[Lynch & Tuttle 88] N. Lynch and M. Tuttle, "An Introduction to Input-Output Automata," Tech Memo MIT/LCS/TM-373, Lab. for Computer Science, MIT, November 1988.

[Ostroff & Wonham 87] J. Ostroff and W. Wonham, "Modelling And Verifying Real-Time Embedded Computer Systems," *Proc. IEEE Real-Time System Symp.*, (December 1987), pp. 124-132.

[Raju 94] S. Raju, "Using Assertions For Validating, Verifying, And Monitoring Real-Time Systems," TR# 94-05-04, Dept. of Computer Science & Engineering, Univ. of Washington, Seattle, May 1994 (Ph.D. Dissertation).

[Raju et al. 92] S. Raju, R. Rajkumar, and F. Jahanian, "Monitoring Timing Constraints in Distributed Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.*, 1991, pp. 57-67.

[Raju & Shaw 94] S. Raju and A. Shaw, "A Prototyping Environment For Specifying, Executing And Checking Communicating Real-Time State Machines," *Software-Practice & Experience* Vol. 24, No. 2, (February 1994), pp. 175-195.

[Shaw 92] A. Shaw, "Communicating Real-Time System Machines," *IEEE Trans. on Software Eng.*, Vol. 18, No. 9, (September 1992), pp. 805-816.

[Shaw 93] A. Shaw, "A (More) Formal Definition of Communicating Real-Time State Machines," TR# 93-08-01, Dept. of Computer Science & Engineering, University of Washington, Seattle, August 1993.

[Shaw 94] A. Shaw, "On Scalable State-Based Specifications For Real-Time Systems," TR# 94-020-3, Dept. of Computer Science & Engineering, Univ. of Washington, Seattle, February 1994.

Towards A Practical Verification Environment for Concurrent Programs *

Robert J. Shaw Ronald A. Olsson Cui Zhang

<lastname>@cs.ucdavis.edu
University of California, Davis

August 24, 1994

1 Introduction

We have recently undertaken the construction of a verification environment for distributed programs. It provides them access to the tremendous rigor and thoroughness of a theorem-prover, but relieves them of the complexity of interacting directly with such a system. Upon completion of this two-year project, we hope to achieve an environment similar to the Programmer's Apprentice [7].

Experience gained from the Davis Silo Project [8], an ongoing formal proof effort to verify a small distributed system in several layers ranging from hardware to concurrent application programs, has shaped our beliefs on the applicability of modern, sophisticated theorem-provers to professional software engineering. We take the viewpoint that the mechanized formal logics of theorem-provers will become the "assembly language" of high-assurance programming, and that tools and environments will be necessary to interact with these logic systems on behalf of the programmer. Within this report, the "high-level" language of interaction is concurrent source code with propositional logic annotations, and possibly, in another mode of conversation, graphical representations of the running program to aid the user's formulation of correct annotations. Through its ability to manipulate the user's decorated source code, to produce the corresponding terms, theorems, and possible proof tactics for the theorem-prover, and to display the prover's results or errors in the high-level form, our proposed verification environment plays the role of a knowledgeable mediator between different levels of abstraction. Our current working title for the system is Snark, that most elusive of beasts.

If correctness proving is to gain widespread acceptance, it must be integrated into the software engineering lifecycle in a manner that allows maintenance and evolution of the proof in tandem with the rest of the system. Source code annotations are a simple means to this integration, and with an intelligent tool such as Snark that tracks dependencies to determine the scope of program changes, that hides unnecessary details through hierarchical organization, and that allows relaxation of rigor for selected portions of code, the proven implementation becomes far more manageable.

The next section describes the Snark system as it appears to the user, and Section 3 sketches our top-level design. Section 4 concludes. Although we will strive for general methods that apply to most theorem-provers and imperative-style languages, these early efforts will use the Cambridge Higher-Order Logic (HOL) system as the underlying formal system [4], and the SR concurrent programming language [2]. Throughout this paper, the term "SR" may mean either the programming language, or the language augmented with annotations. Likewise, the terms "HOL" and "theorem-prover" will refer to either the mathematical formal system or to the software system which mechanizes it. These meanings will be clear in context.

* This work is sponsored in part by the US Department of Defense University Research Program

2 System Description

2.1 The Underlying Concurrent Programming Logic

Hoare-style programming logics [6], and their syntactic shorthand known as proof outlines have long been studied, and more recently, embedded within mechanized logic systems such as HOL [9]. To fully decorate a concurrent program with first-order logic terms so that it represents a completely formal derivation within the Hoare logic is much simpler than reasoning directly from the specification of the language semantics within HOL. Indeed, with sequential code, Hoare logic proofs can be largely automated, up to the selection of loop invariants, but concurrency introduces new complexities into the Hoare logic approach. Yet, by carefully choosing axioms and inference rules that require the programmer to state properties in an orderly fashion, the potential for automation and for intelligent assistance increases. We hope that an orderly Hoare system yields underlying HOL proofs with a predictable structure, albeit with a greater number of steps as well. However, when the low-level proofs are not read by people, it is not a problem if the ultimate number of HOL proof steps grows in order to gain the modularity that accommodates manipulation by a software system. True proof maintenance, a feature which is crucial to long-term software projects, may be facilitated through this approach.

In our programming logic, as in [1], the SR programmer views each communication channel as modeled by a list object, and a prefix of this list which respectively represent all the messages sent into the channel from a given process and those that have been received thus far in the computation. Our logic requires an invariant property on each channel, describing the nature of the communication along that link. Such a policy is good software engineering anyway, and would likely interface well with software tools for producing design documentation and perhaps requirements acquisition as well, such as recent work at UBC [5], but its primary purpose is to shape the overall proof into a predictable structure. Expressive power is not lost by this requirement, but experiments by hand indicate that automated deduction potential is gained. Indeed, existing HOL libraries for arithmetic, and more recently for lists, have incorporated this same notion of a preferred phrasing for automatic proving. In Snark, we suspect it best to formulate channel properties as a conjunction of implications, such as the following illustration:

```
(IF <in critical section> And <request is READ>  THEN <no writers>) And
(IF <in critical section> And <request is WRITE> THEN <....>) And
(IF <buffer full> THEN <not in critical section>) And
  . . . .
```

Not only is this style a likely mirror of the programmer's thinking, but it aids mechanical resolution as well.

Taken together, all the channel invariants form a systemwide property that must hold everywhere, and this particular predicate plays a central role in the proof of a distributed program. Within Snark, it facilitates:

1. **Non-Interference Proofs.** As a side condition to the Hoare rule that composes multiple processes, there is a proof obligation which amounts to showing that any statement that might be running in a foreign process can not falsify local assertions. When assertions are completely disjoint these theorems are trivial, in other cases, since the global invariant must always be true, resolving it against the local assertion should yield proof insights, if not a valid proof.

2. **Modularity.** Without shared data, processes may only communicate through channels. If the proof of a particular process doesn't (and in good code, it shouldn't) rely upon the status of foreign local variables, then the invariant encapsulates completely how the process interacts with the rest of the system. Any internal change to the process which conforms to the invariant would only require local re-proving.

2.2 Modes of Use

Since this prototype is a general-purpose tool for manipulating annotated distributed programs, there are several modes of operation available to the user. From our manual proofs of tiny Silo programs, three common paradigms are: verification of communication activity, high-level analysis of composed system components, and complete proof of critical processes and code fragments.

Verification of communication activity is an iterative, exploratory approach that stems from the nature of our concurrent programming logic. The goal is to find an appropriate characterization of the system's interprocess communication that is suitable for proof purposes. As previously shown, this invariant typically contains several conjuncts, each of which is an implication depending upon program variables such as semaphores or counters.

Just as a well-designed distributed system should exhibit modularity and delegation of responsibility, so too will these attributes be apparent within the invariant assertion – there will likely be distinct subsets of conjuncts which describe disjoint components of the system being verified. Once the user believes a correct invariant is in hand, the first step is to ensure that it is maintained across all program statements. In many cases, disjoint variables, false antecedents, and proof fragments from earlier iterations of attempted invariants will partially prove and simplify the invariant differently at different locations in the source code. The programmer examines which conjuncts remain unproven, and decides to suggest Hoare rules, or to modify code and annotations as required. An inappropriate invariant will be exposed by the conjuncts which remain and where in the program they reside. Successful proving of remaining conjuncts can likely be repeated everywhere in the code that is not modifying the structures described by those conjuncts.

High-level analysis is the viewing of entire processes as black boxes, and examining relationships between their input and output streams, and their effects on the invariant. Here, the starting point is not an attempt at a correct predicate, but rather, the asserted relationships for the output of a process in terms of its inputs. Since the prototype allows this kind of assumption, the user may explore high-level aspects of system integration by simply asserting that a particular process behaves correctly. An example is perhaps three processes all acting as filters – what facts are true of the final output given various properties of the originating stream of messages?

Complete proving of crucial code is much like traditional verification. The user begins with initial conditions for the code section, and then attempts to annotate every line. When interaction with other processes is not considered, the normally required global invariant may be taken simply as the predicate *true*, allowing the user to concentrate upon local properties. The main burden on the programmer is to find the appropriate loop invariants, as the prototype can often carry an assertion through straight-line code automatically. When the programmer makes alterations such as strengthening part of a loop invariant from `x > 0` to `x > 1`, the prototype attempts to reprove the outline with the appropriate substitutions. In turn, this action will direct the user to previous assertions or to source code that must change to accommodate the new outline, and the whole process proceeds iteratively.

3 System Components

Since the primary goal of our system is to raise the level of discourse between programmer and theorem-prover, it is not surprising that our proposed system will have three main components: an interactive front-end system, an intermediate liaison, and the theorem-prover itself. Figure 1 depicts this design as described below. Space permits only a brief mention of selected Snark features.

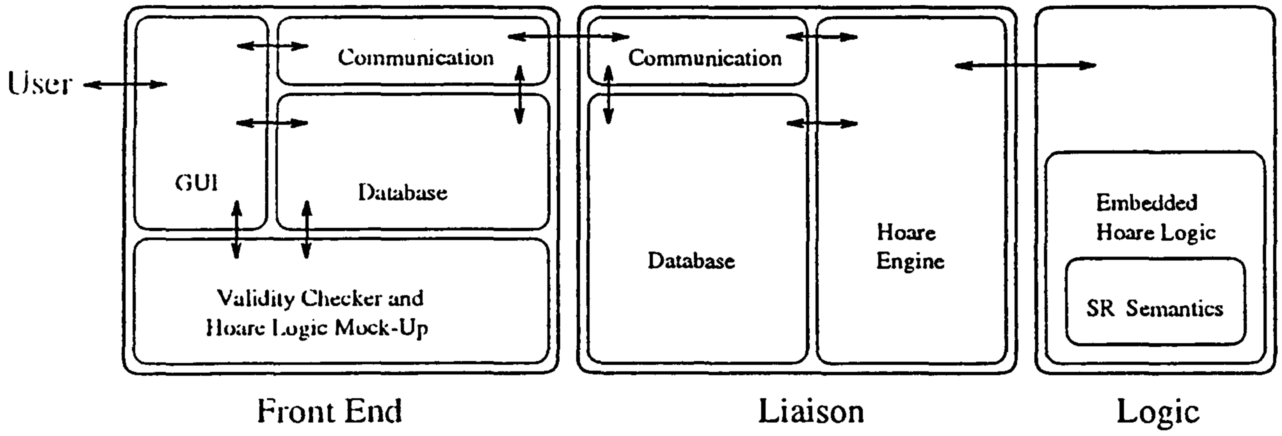


Figure 1: Design Sketch of the Snark system

3.1 Front-End System

For interaction, the front-end provides a syntax-directed GUI editor in which the user edits source code and annotations, each hierarchically structured so that large portions may be given single identifier aliases. Together, statements and annotations form the familiar "triples" of a Hoare logic, {P} S {Q}, where the block S is recursively composed of smaller triples. The rules for well-formed proof outlines unambiguously dictate which Hoare axiom or inference rule is indicated and which triples are involved at each proof step for completely decorated code. In the prototype, decorating code whose semantics are not formalized (e.g. floating-point expressions) is disallowed.

Prominent features and responsibilities of the front-end are as follows:

1. **Validity Checks.** Scoping, typing, and other relations between annotations and neighboring source statements are always ensured. Using aliases and cut-and-paste would quickly introduce errors without this facility. By preventing ill-formed assertions and triples, Snark prevents certain erroneous code from ever going to the theorem-prover.

2. **Progress Management and Memoization.** As the user constructs program triples and other entities, the front-end manages these objects and their hierarchical substructures. The liaison returns object tags for communication purposes between these two components. More importantly, an association is maintained between the triples, and the actual proof tactics that created them within the theorem-prover, so that small changes will only require re-proving of the affected objects.

3. **Other Assistance.** When stand-alone theorems are required by the hypothesis of a Hoare rule, browsing of possible candidates from libraries and existing user theorems is provided. The user may also convert assertions to the preferred phrasing automatically, and "execute" them to determine which system configurations (constructed graphically) are admitted or precluded by a particular logical term. Finally some translation must occur before communicating with the liaison, such as qualifying a user's "x" variable with scoping information.

3.2 Liaison

This component services proof requests from the front-end by translating them into the appropriate form for the underlying theorem-prover, and attempting to run them. Hoare inference rules for generating the user's triples are really just theorems in the underlying prover, as are the triples themselves. That a set of program triples, as HOL theorems, really does meet the hypothesis of an inference rule, as a HOL theorem, needs to be rigorously proven. It is the Hoare Engine – the brain of the liaison – which translates the user's/front-end's Hoare derivations into HOL tactics. Moreover, the Hoare Engine keeps dependencies amongst all the triples that it creates. Should the user change an assertion or a portion of code, the Engine can retrace the forward Hoare proof to determine not only what triples are invalidated, but also what subterms of those triples are responsible. As previously mentioned, the Engine provides tags for these objects back to the front-end, which informs the user of how the proof is affected. Simply substitutions such as x > 0 becoming x > 1 will hopefully not confound the Engine, assuming an already known proof remains valid when the steps pertaining to the first case are adjusted mechanically.

The Hoare Engine has both libraries of common proof idioms (an index stays within a range, a communication channel remains ordered, the set of receivers for a given channel contains only one process), as well application-specific information (two arms of an if differ at only one statement) that allow it to rework proof derivations within the programming logic in an intelligent manner, discovering trouble spots. Moreover, in many cases it can store similar information about the low-level proof tactics as well, allowing the analysis of the postcondition weakening discussed above. When side conditions are necessary to invoke a Hoare rule, the Engine will generate them as theorems or goals.

Finally, because of the presence of this Engine, the user is free to adjust the level of rigor at various locations in the program. At the highest level, the Engine is compelled to produce a true theorem representing the program triple within the underlying logic before reporting success. On the other hand, the user may also request that certain triples are simply assumed to be true, in order to expedite the proving of more crucial portions of code.

3.3 Theorem-Prover Support

Naturally, the theorem-proving system itself must contain a mechanization of the programming logic, which in turn requires embedded theories for the syntax and semantics of the programming language. Having succeeded in a similar task for Silo, we will produce a HOL embedding of our logic which is amenable to manipulation by the Hoare Engine. Also, we will prove certain properties of our semantic specification that demonstrate its conformity to our informal notion of SR semantics.

4 Conclusions

Armed with modern theorem-proving technology — systems for which a myriad of tiny details and complex mathematical abstractions are no more daunting than 1+1=2 — we feel that the raw power to formalize concurrent systems is well in hand. Projects such as the Davis Silo and the CLI work with kernels [3] support this claim. Transferring this technology to the programmer so that the complexity becomes manageable and maintainable is now the challenge before us.

If we succeed, our contributions to practical software engineering would be primarily these:

- **Complexity Reduction** Just as an SR implementation is far more maintainable than raw MIPS code, so too must veriliers work in a language much higher than raw logical inferences. Furthermore, if that language models the programmer's own thinking, as we have attempted, the discourse is all the more fluent.
- **Modularity**. With Snark, separate system components can be proven separately, since a well-crafted invariant encapsulates each process' interaction with the rest of the system

Modern theorem provers do not yet encompass these needs of realistic software projects, and it is our claim that they shouldn't attempt to do so. Just as we have compilers and debuggers, we will need a third family of tools pertaining to verification for those who seek the high-assurance that formal methods can provide. Still in our first month, we have only begun exploring possible implementation strategies for Snark, but have encountered no major difficulties thus far.

References

[1] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.

[2] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Co., Redwood City, CA., 1993.

[3] W. Bevier and L. Smith. A mathematical model of the Mach kernel: atomic actions and locks. Technical Report 89, Computational Logic, Inc., Austin, TX, April 1993.

[4] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higer Order Logic*. Cambridge University Press, 1993.

[5] J. Joyce, N. Day, and M. Donat. S: A machine-readable specification notation based on higher order logic. In *HOL 1994: International Conference on Higher Order Logic Theorem Proving and its Applications*, to appear, September 1994.

[6] S.S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279 285, May 1976.

[7] C. Rich and R. Waters. *The Programmer's Apprentice*. Addison-Wesley, 1990.

[8] C. Zhang, R. Shaw, M. R. Heckman, G. D. Benson, M. Archer, K. Levitt, and R. A. Olsson. Towards a formal verification of a secure distributed system and its applications. In *The 17th National Computer Security Conference*, to appear, October 1994.

[9] C. Zhang, R. Shaw, R. Olsson, K. Levitt, M. Archer, M. Heckman, and G. Benson. Mechanizing a programming logic for the concurrent programming language microsr in HOL. In *Higher Order Logic theorem Proving and Its Application*, number 780 in Lecture Notes in Computer Science, pages 29–42. Springer-Verlag, 1994.

Formal Methods and Software Maintenance — Some Experience with the REFORM Project

Hongji Yang
Computer Science Department
De Montfort University
England
email: hjy@uk.ac.dmu
fax: +44 (0) 533 541 891

1 Introduction

Formal methods could play a more prominent role in enhancing software quality in the software life cycle. Because software maintenance is becoming an increasingly important phase of the software life cycle, software maintenance using formal methods is an issue that should be addressed promptly by researchers. The REFORM project was such a project, and within the REFORM project, formal methods were used as much as possible. This paper presents experience and reflections, obtained from carrying out the REFORM project, on the application of formal methods to software maintenance.

2 REFORM Project

REFORM - Reverse Engineering using FORmal Methods - was a joint project between University of Durham, CSM Ltd. and IBM (UK) to develop a tool called the Maintainer's Assistant. The REFORM project started in July 1989 [6] and finished in March 1993. The aim of the project was to build a prototype tool — the Maintainer's Assistant — which would take existing software written in low-level procedural languages (in particular, IBM CICS code written in IBM-370 assembler), through a process of successive transformation, and turn it into an equivalent high-level abstract specification expressed in terms of a non-procedural abstract specification language (in particular, Z). Naturally, as the process of applying program transformations cannot be totally automated, the Maintainer's Assistant is an interactive tool including an interactive interface.

Theoretical foundation The REFORM Project has its roots in Ward's work [4] in which he developed methods of proving refinements and transformations of programs. Although he used the popular approach of defining a core "kernel" language with denotational semantics, and permitting definitional extensions in terms of the basic constructs, he did not use a purely applicative kernel. Instead, the concept of states is included, using a *specification statement* which also allows specification expressed in first order logic as part of the language (thus providing a genuine wide spectrum language).

Using Ward's approach it is possible to prove that two versions of a program are equivalent. Programs are defined to be equivalent if they have the same semantic function. Hence equivalent programs are identical in terms of their input-output behaviour, although they may have different running times and use different internal data structures. A refinement of a program, or specification, is another program which will terminate on each initial state for which the first program terminates, and will terminate in one of the possible final states for the first program. In other words a refinement of a specification is an acceptable implementation of the specification and a refinement of a program is an acceptable substitute for the program.

Ward's work can be used in both program development and software maintenance.

The Wide Spectrum Language In the process (within the REFORM project) of acquiring a specification from the program code, a notation (or a language) is needed to represent the program and specification at all levels, especially as objects (program or specification) are changed from one form to another. A wide spectrum language suitable for this, named WSL, has been defined, which incorporates a variety of constructs, from low-level machine-oriented constructs up to high-level specification ones.

WSL [3,4] consists of two types of construct: program-specification WSL constructs (for representing both program code and program specification) and Meta-WSL constructs (for representing program transformations). Both types of WSL constructs originate from the kernel language.

The Maintainer's Assistant The Maintainer's Assistant [7] works like this: a "Source-to-WSL" translator takes the assembler (or other language) and translates it into its equivalent WSL. The maintainer undertakes all operations through the Browser. The Browser will check the program, chop the program into smaller programs which are of manageable size, and save it in a database.

Then, the maintainer will take a piece of code from the database to work on. The Browser allows the maintainer to look at and alter the code under strict conditions and the maintainer can also select transformations to apply to the code. The program transformer works in an interactive mode. It presents WSL on screen in a pretty printed format and searches a catalogue of proven transformations to find applicable transformations for any selected piece of code. These are displayed in the user interface window system. When the Program Transformer is working, it also depends on the General Simplifier, the Program Structure Database and the Metric Facility by sending them requests. Once a transformation is selected it is automatically applied. These transformations can be used to simplify code and expose errors. Finally, the code is transformed to a form at a higher level of abstraction, which can be translated into specifications in Z (Figure 1).

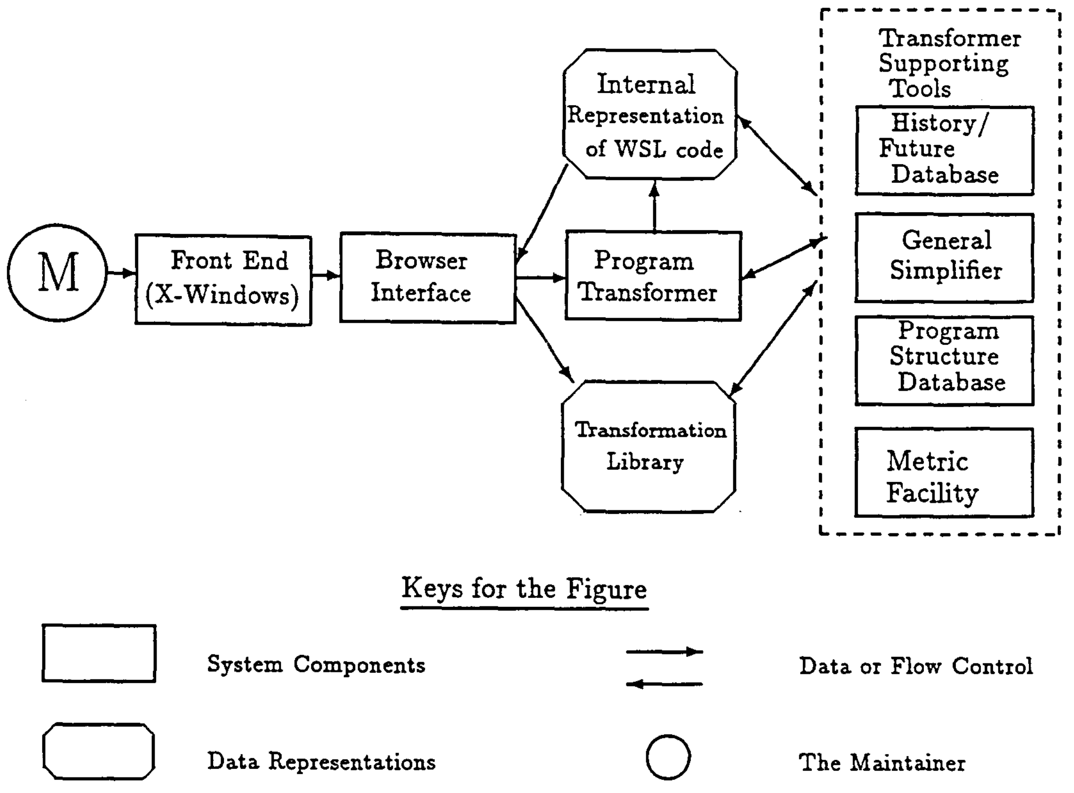


Figure 1: Main Components of the Maintainer's Assistant

Results Part of the REFORM project has involved taking source code written in IBM 370 Assembler and using the system on it [1]. In order to do this, it is necessary first to translate the Assembler into WSL. Once the code has been translated into WSL, it is then possible to use the Maintainer's Assistant to remove all the extra, unnecessary code that was introduced by the translation process and to do some simple tidying of the code. Once the code is in a tidier form the maintainer can then work on it, simplifying and restructuring it. Doing this has yielded further improvements to the code through the identification and production of loops, procedures and data structures. The tool has been used on a selection of systems and applications written in IBM 370 Assembler. In all instances, a maintainer who was unfamiliar with the particular modules was responsible for working on the code and in all instances very significant improvements were made, not only in the structure of the code but also in its comprehensibility.

One of the case studies [5] completed in the REFORM project involved a number of modules of IBM Assembler, each consisting of up to 20,000 lines of code, taken from a large commercial system. Each module was automatically translated into WSL and interactively restructured into a high-level language form. One particular module had been repeatedly modified over a period of many years until the control flow structure had become highly convoluted. Using the prototype tool we were able to transform this into a hierarchy of (single-entry, single-exit) subroutines resulting in a module which was slightly shorter and considerably easier to read and maintain. The transformed version was hand-translated back into Assembler which (after fixing a single mis-translated instruction) "worked first time". A typical result (measured by the metrics discussed in [1]) for one of the smaller modules is shown in the table below:

Stage	lines	McCabe	Structural	Size
Translated WSL code	2,330	1,030	48,175	24,736
After automatic restructuring	1,381	245	17,021	8,404
After small amount transformations	1,227	156	11,990	7,120

Programs written in COBOL were also addressed by the REFORM project [8]. The aim was to acquire program designs from code written in COBOL. These program designs were represented in Entity-Relationship Attribute Diagrams, as COBOL programs are often designed using Entity-Relationship Attribute Diagrams, rather than process based design methods.

3 Comments

The REFORM project has been highly successful (largely as a result of using formal methods), producing a reverse-engineering tool, which is already capable of producing useful results for real "spaghetti" assembly modules and real COBOL modules.

The REFORM approach used formal program transformations and the benefits of which are:

- Increased reliability: bugs and inconsistencies are easier to spot.
- Formal links between specification and code can be maintained.
- Maintenance can be carried out at the specification level.
- Large restructuring changes can be made to a program with the confidence that the functionality is unchanged.

To reduce the degree of reliance on the skill required by the maintainer who uses the Maintainer's Assistant, the Program Transformer checks the applicability of a transformation and only applicable transformations can be applied.

We tried to use existing tools constructed by formal methods to speed up building the Maintainer's Assistant. For example, the Boyer-Moore Theorem Prover (BMTP) [2] was used to construct part of the General Simplifier. It was found not only that a lot of work needs to be done on interfacing the Maintainer's Assistant to the BMTP but also that the BMTP is not really powerful enough to provide the service.

Although the original aim of the project was simply to investigate the application of formal methods to reverse engineering, efforts were made to use formal methods throughout the entire software maintenance process. This was achieved by providing formal transformations for transforming specifications or designs to low level code. It means when a specification or design was obtained from old code, new code can be generated from the obtained specifications by program transformations.

Some of the main components of the Maintainer's Assistant, such as most of the transformations, the Program Structure Database and the Metric Facility, were implemented with the help of using formal methods. In these cases, formal specifications for those components were written first in WSL. It was found that the coding for these components was straightforward and the reliability of these components was high.

However, the number of formal transformations implemented in the system is crucial to the success of the system, in particular, when a large program is involved. It was also found that the Maintainer's Assistant was able to cope with a program of several hundred lines well but more supporting tools, such as a Slicer, were needed when dealing with programs of a few thousand lines. Furthermore, the response time of the prototype becomes longer as the size of the program increases.

It is noted that the Maintainer's Assistant has some limitations and therefore extensions to the tool are needed: firstly, an extension to include more high-level transformations to produce abstract specifications from code; secondly, an extension of the theory to communicating parallel programs; thirdly, extensions to deal with real-time and interrupt-driven programs.

In conclusion, formal methods seem to represent an area of tension between industry and academia. Academics are hailing formal methods as the best solution to many problems which exist in the software industry, but industry remains very sceptical. It has to be recognised that formal methods are at very early stage of industrialisation, and a considerable amount of software engineering work is necessary before they can be brought to everyday use. However, formal methods hold out much hope for improving software maintenance. The experience gained with the REFORM project showed that formal methods offer the possibility of maintaining a specification, rather than maintaining code itself. This means that formal methods may allow us to maintain systems at a much higher level of abstraction than we are able to do currently.

References

[1] Bennett, K. H., Bull, T. and Yang, H., "A Transformation System for Maintenance — Turning Theory into Practice", IEEE Conference on Software Maintenance-1992, Orlando, Florida, November 1992.

[2] Boyer, R. S. and Moore, J. S., *A Computational Logic*, Academic Press, Inc., New York, 1979.

[3] Bull, T., "An Introduction to the WSL Program Transformer", IEEE Conference on Software Maintenance-1990, San Diego, California, 1990.

[4] Ward, M., "Proving Program Refinements and Transformations", Ph.D. Thesis, Oxford University, 1989.

[5] Ward, M. and Bennett, K. H., "A Practical Program Transformation System for Reverse Engineering", IEEE Conference on Software Maintenance-1993, Montreal, Canada, 1993.

[6] Ward, M., Munro, M. and Calliss, F. W., "The Maintainer's Assistant", IEEE Conference on Software Maintenance-1989, Miami, Florida, 1989.

[7] Yang, H., "The Supporting Environment for A Reverse Engineering System — The Maintainer's Assistant", IEEE Conference on Software Maintenance-1991, Sorrento, Italy, October 1991.

[8] Yang, H. and Bennett, K. H., "Extension of A Transformation System for Maintenance — Dealing With Data-Intensive Programs", IEEE International Conference on Software Maintenance (ICSM '94), Victoria, Canada, September 1994.

A SOFTWARE EVOLUTION CONTROL MODEL

Salah Badr
Information Systems Branch
Egyptian Armament Authority
Cairo- Egypt

Valdis Berzins
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943 USA
berzins@cs.nps.navy.mil

ABSTRACT

This paper presents an evolution control system that provides automated assistance for the software evolution process in an uncertain environment where designer tasks and their properties are always changing.

We view an Evolution Control System (ECS) as the agent that keeps track of proposed, ongoing, and completed changes to a software system. It provides automated assistance to the software evolution manager to help him/her make the right decisions. It automatically propagates change consequences by constructing the set of possibly affected modules. It also coordinates change implementation activities within the design team in a way that supports team work and maintains system integrity, as well as adapting itself to the dynamic nature of the evolution process where new changes arrive randomly and ongoing modifications are themselves subject to change as more information becomes available.

A. INTRODUCTION

An ECS has two main functions. The first is to control and manage evolving software system components (version control and configuration management, VCCM) and the second is to control and coordinate evolution team interactions in a way that maximizes the concurrent assignment and meets management constraints such as deadlines and precedences (planning and scheduling software evolution tasks which we refer to as evolution steps).

This system provides the required algorithms for coordinating and executing the activities mentioned above as well as the algorithms for reaching and maintaining a feasible schedule, if one exists, that meets the deadline requirements, reduces/avoids rollbacks, and insures system integrity in an uncertain environment where the set of evolution steps and their properties are always changing.

B. CONCEPTUAL MODEL

Since the main purpose of the ECS is managing software evolution in a rapidly evolving system, we review the graph model of software evolution that constitutes the context for building the ECS [6] [7]. The goal of this model is to provide a framework for integrating software evolution activities with configuration control [6]. The model of software evolution has two main elements: system components and evolution steps. System components are immutable versions of software source objects that cannot be reconstructed automatically. Evolution steps are changes to system components that have the following properties in the original version of the graph model [6]:

1. A top-level evolution step represents the activities of initiation, analysis, and implementation of one change request.
2. An evolution step may be either atomic or composite.
3. An atomic step produces at most one new version of a system component. This property is no longer true in our model in order to include the cases in which an atomic step is applied to an originally atomic component that needs to be decomposed according to some design considerations. This decomposition may lead to the production of more than one component.This modification is illustrated in section C.2.e later in this chapter.
4. The inputs and outputs of a composite step correspond to the inputs and outputs of its substeps.
5. The model allows steps that do not lead to the production of new configurations, e.g. design alternatives that were explored but not included in the configuration repository.
6. Completely automatic transformations are not considered to be steps and are not considered in this model.
7. The graph model can cover multiple systems which share components, alternative variations of a single system, and a series of configurations representing the evolution history of each alternative variation of a system.
8. A scope is associated with each evolution step which identifies the set of systems and variations to be affected by the step. The scope is used to determine which induced evolution steps are implied by a change request.

The evolution history is modeled as a graph G=[C, S, CE, SE, I, O]. This graph is a directed acyclic graph (bipartite with respect to the edges I and O). C and S are the two kinds of nodes (C: software component nodes, and S: evolution step nodes respectively). Each node has a unique identifier. C and S nodes alternate in each path that has only I and O edges. This represents the evolution history view of the graph. The edges represent the "part_of" (between a sub-component of a composite component and the composite component) and "used_by" relations (defined between components to represent the situation where the semantics or implementation of one component A depends on another component B; B used_by A) between the software components of a given configuration ($CE \subseteq C \times C$), the "part_of" relation between a substep of a composite step and the composite step ($SE \subseteq S \times S$), the input relation between the system components which must be examined to produce output components that are consistent with the rest of the system and the corresponding evolution steps($I \subseteq C \times S$), and output relation between evolution steps and the components they produce ($O \subseteq S \times C$). System components are immutable versions of software source objects that cannot be reconstructed automatically.

An "edge_type" attribute is used to distinguish between the two kinds of edges representing the relations "used_by" and "part_of" defined on the set of edges $CE \subseteq C \times C$. The "used_by" relation can be used for automatic identification of inputs of proposed evolution steps and identification of the induced steps triggered by a proposed step.

The model distinguishes between the primary and secondary inputs of a step. The primary input concept can be formalized by introducing the attributes object_id, version_id and variation_id of each version. Variations represent alternative choices, which may correspond to different formulations of the requirements in the context of prototyping, or different kinds of system software (operating system, window manager, etc.) in the context of product releases. Each variation is a linearly ordered sequence of versions. An input to a step is primary if and only if it is the previous version of the same object and belongs to the same variation as the output of the step.

1. Version and Variation Numbering

As soon as the input base version of a step is bound, the system assigns the version and variation number of the output object for the step. The variations are assigned successive numbers beginning with 1 for the initial variation. Versions along each variation are assigned successive numbers starting with 1 at the root version of the initial variation. This means that the new version number is the base version number plus one, while the variation number has two possibilities: the first possibility is to keep the base version's variation number at the time the step is assigned. This occurs when the base version is the most recent version on its variation line at the time the step is assigned. The other possibility is to use the "next" variation number, which is the highest variation number plus one. This labeling function illustrated in Figure 1 is the same for both atomic or composite objects (the entire software system is represented as a composite object).

This labeling function allows a version to belong to more than one variation which is a necessary modification to [6] to simplify the process of tracing the development history of a version and to keep a logical and realistic development history.

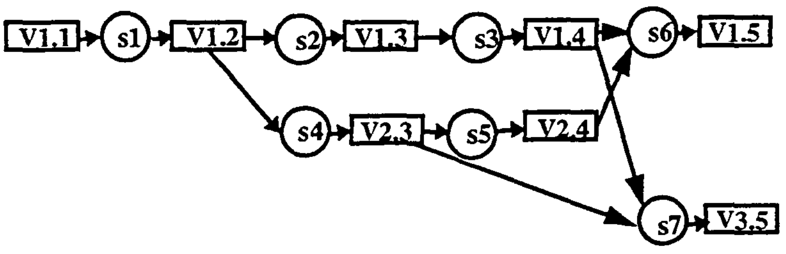


FIGURE 1. Variation and version numbering

2. States of Evolution Steps

The dynamics of the evolution steps are modeled by associating six different states with each step to express the different activities each step has to undergo during its lifetime. The state transition diagram in Figure 2 shows the different explicit decisions that have to be made by the management to cause the transition from one state to the other. It also shows the automated transitions from the scheduled state to the assigned state and vice versa (explained in detail in subsections c, and d below). By controlling the states of the evolution steps, the evolution manager exercises direct control over both software evolution/development and the resulting software configurations. The following are the definitions of those states and the corresponding actions that cause the transition from one state to the other. These states are similar to those presented in [6] except that a new state called "assigned" has been added for the reasons explained below.

a. Proposed State

In this state a proposed evolution step is subjected to both cost and benefit analysis. This analysis also includes identifying the software objects comprising the input set of the step. A "proposed" step is generally added to the configuration graph as an isolated step node that does not have any input, output or part_of edges (except when an old version is used that has existing specific reference). This is because the primary and secondary input attributes are mostly generic inputs (object_id and variation_id only).

b. Approved State

In this state the implementation of the step has been approved but not scheduled yet and the input set of the step is not bound to particular versions. Approval of a proposed step by the management triggers the decomposition process to create an atomic sub-step for each primary or affected component of the step. These sub-steps inherit the status of their super-step which is "approved" in this case, and are added to the configuration graph with a part_of edge between each sub-step and its super-step. It is also in this state that the substeps are augmented with attributes that include the estimated duration of each sub-step and management scheduling constraints such as precedence, deadline, and priority.

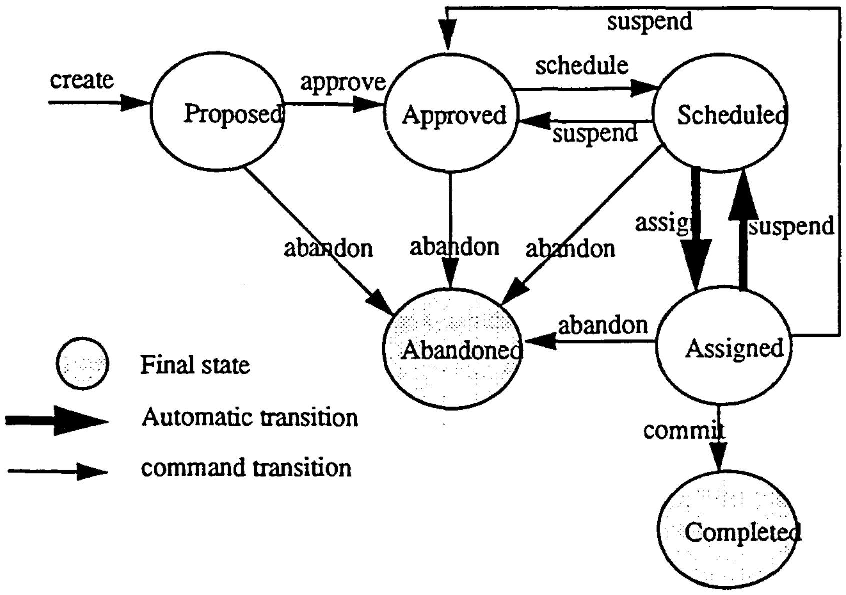


FIGURE 2. Evolution step's state transition diagram

c. Scheduled State

In this state the implementation has been scheduled and the step is not yet assigned to a designer. The "scheduled" state is reached from the "approved" state via the command "schedule_step" that indicates that the management constraints are complete and enables the scheduling and job assignment mechanisms. The scheduling mechanism produces an updated schedule containing the newly scheduled step. A schedule specifies the expected starting and completion times for the step.

d. Assigned State

In this state the step is assigned to the scheduled designer, all inputs are bound to particular versions, and unique identifiers have been assigned to its output components, but these components are not yet part of the evolution history graph. A composite step enters the assigned state whenever any of its substeps is assigned.

The assigned state is reached automatically from the scheduled state. When a designer is available, the schedule is used to determine his/her next assignment. If his/her next assignment is ready to be carried out then the step status is automatically advanced to "assigned" and the designer is informed of the new assignment. When a step is assigned, the version bindings of its inputs are automatically changed from generic to specific. An edge is added as an input edge between the primary input component of the step and the step itself in the configuration graph.

e. Completed State

In this state the outputs of the step have been verified, integrated, and approved for release. This is the final state for each successfully completed step. This state can only be reached from the assigned state using the "commit_step" command. In this state the output components of the step have been added to the configuration graph. An output edge has also been added to the configuration graph between the step and its output component(s). A composite step enters the completed state when all of its substeps are completed

f. Abandoned State

In this state the step has been cancelled before it has been completed. The outputs of the step do not appear as components in the evolution history graph. All partial results of the step and the reasons why the step is abandoned are stored as attributes of the step for future reference. This is the final state for all steps that were not approved by the management or cancelled in the "approved", "scheduled" or "assigned" states.

3. SCHEDULING MODEL

The task in our case is to schedule a set of N evolution steps $S = \{S_1, S_2, \dots, S_N\}$ relative to a set of M designers $D = \{D_1, D_2, \dots, D_M\}$. The designers are of three possible expertise levels {Low, Medium, High}. Each step has associated with it a processing time tp (S_i), a deadline d (S_i), a priority p (S_i), and required expertise level e (S_i). Steps have precedence constraints given in the form of a directed acyclic graph $G = (S, E)$ such that $(S_i, S_j) \in E$ implies that S_j cannot start until S_i has completed.

Because of the dynamics of the prototyping/evolution process, the steps to be scheduled are only partially known. Time required, the set of sub-tasks for each step, and the input/output constraints between steps are all uncertain, and are all subject to change as evolution steps are carried out.

Our goal is to dynamically determine whether a schedule (the time periods) for executing a set of evolution steps exists such that the timing, precedence, and resource constraints are satisfied, and to calculate this schedule if it exists.

C. DESIGN

The purpose of the Evolution Control System, ECS, is to provide automated support for changes in plan during the execution of the plan, and provide automatic decision support for planning and team coordination based on design dependencies captured in the configuration model. The ECS also manages the software evolution steps from its creation to completion and provides automatic version control and configuration management for the products of these steps.

a. Context Model

The Evolution Control System (ECS) interacts with two external entities: the software evolution manager and the software designer. These represent classes of human users rather than external software or hardware systems. There is one external interface for each class of user: the manager_interface and the designer_interface. Both of these interfaces are views of the proposed ECS. The message flow diagram in Figure 3 and the stimulus-response diagrams in

Figures 7, 8, 9 and 10 show the context of the system and the available commands, their effects and the possible error conditions.

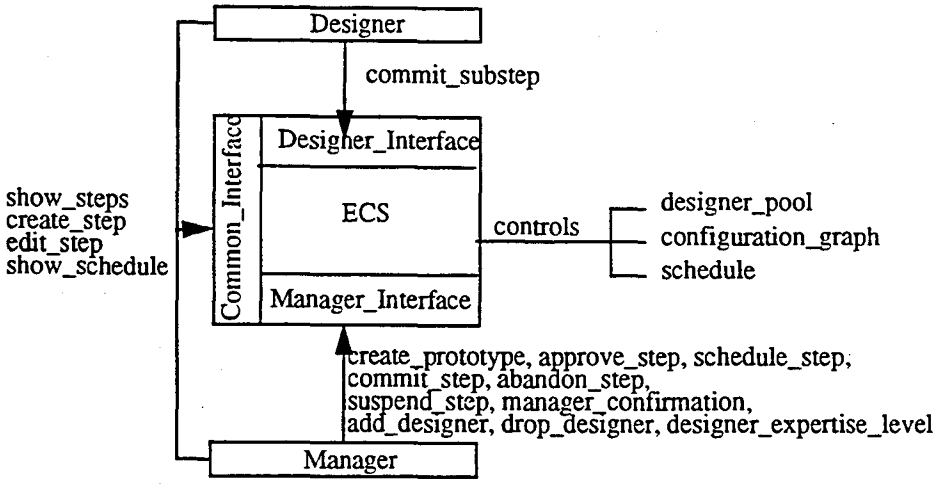


FIGURE 3. ECS message flow diagram

1. State Model And Related Concepts

The state of the ECS consists of a configuration graph, a schedule, a set of designers, and mappings giving the following attributes for each evolution step: deadline, estimated duration, precedence, priority, status and required expertise level. The formal definitions of the state model and the constraints on a feasible schedule are defined in [1].

2. Interfaces

The manager interface to the ECS enables the manager to create new prototypes, provide for the evolution of the existing prototypes via a complete set of commands for creating, editing, scheduling, suspending/abandoning and/or committing evolution steps, and manage the designer_pool data via add_designer, drop_designer, and designer_expertise_level commands. The designer interface to the ECS enables the designer to view the steps in a given prototype with a given status and get the sub-steps assigned to him. This interface also enables the designer to create a sub-step of an assigned step as well as committing the assigned sub-step.The formal specifications of the various commands with the different responses for each command are defined in [1].

The following parameters can be adjusted manually (using the edit_interface) as uncertainties are resolved and planning errors are corrected. 1. Affected modules (Add/del). 2. Secondary input (Add/del). 3. Constraints (Precedence, Priority, Deadlines) (Initialize/Update). 4. Estimated duration (Update). 5. Resource (Designer Pool Changes) (Add/drop, Update).

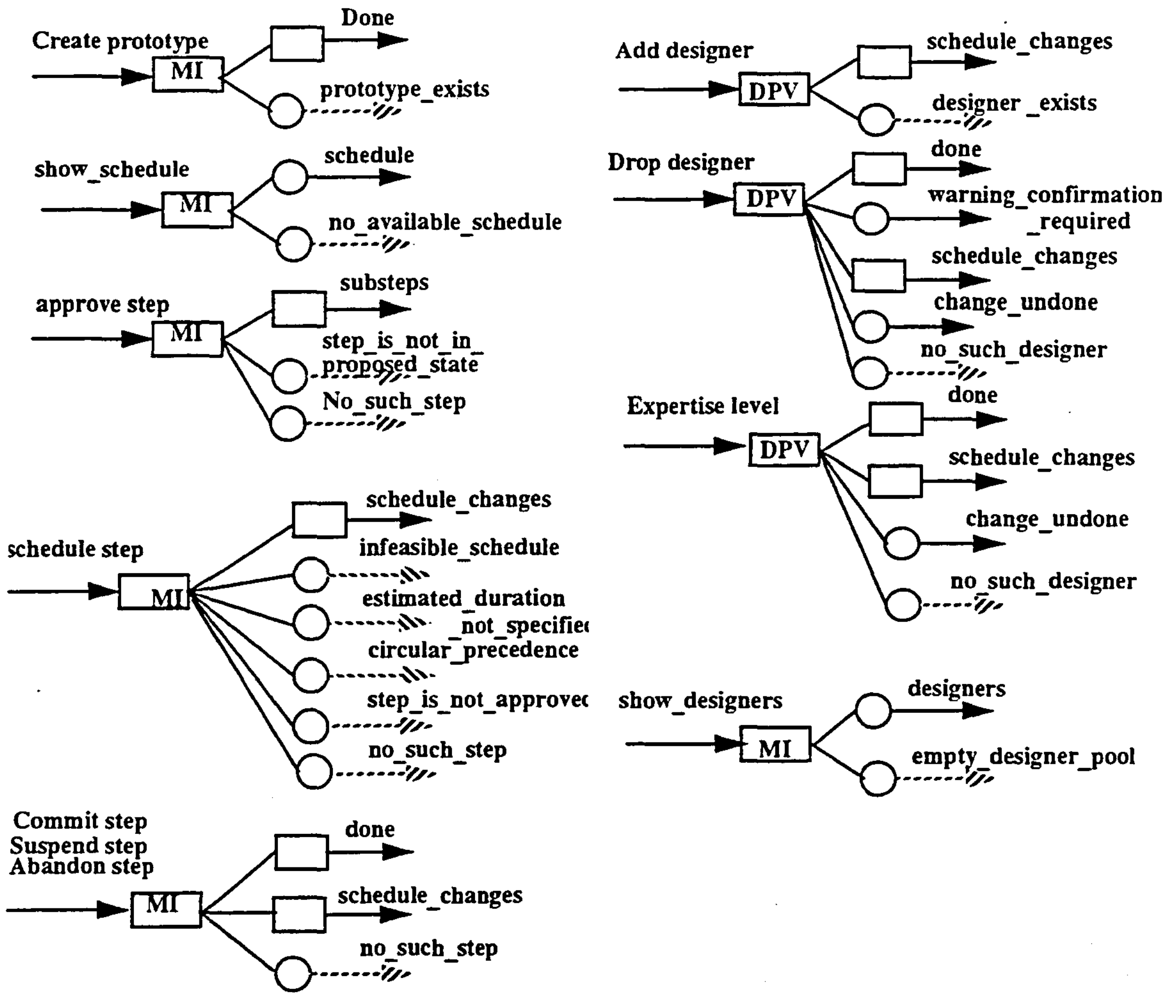


FIGURE 4. Stimulus Response diagram for the manager interface

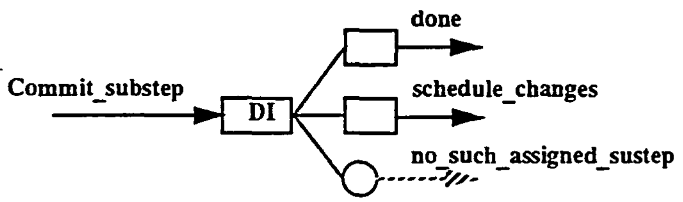


FIGURE 5. Stimulus-Response diagram for the designer interface

The schedule_step command triggers the scheduling mechanism that finds a feasible schedule if one exist or suggest changes to the deadlines of the lower priority steps until a feasible schedule is reached. When a designer is available for his assignment the ECS automatically checks out the required components from the design database to the designer's workspace and sends an e_mail message to the designer informing him about his new assignment. When a designer finishes his assignment, he simply issues the commit_step command. The system then automatically checks in the modified components to the design database giving them the right version and variation numbers and binding them to the appropriate configuration.

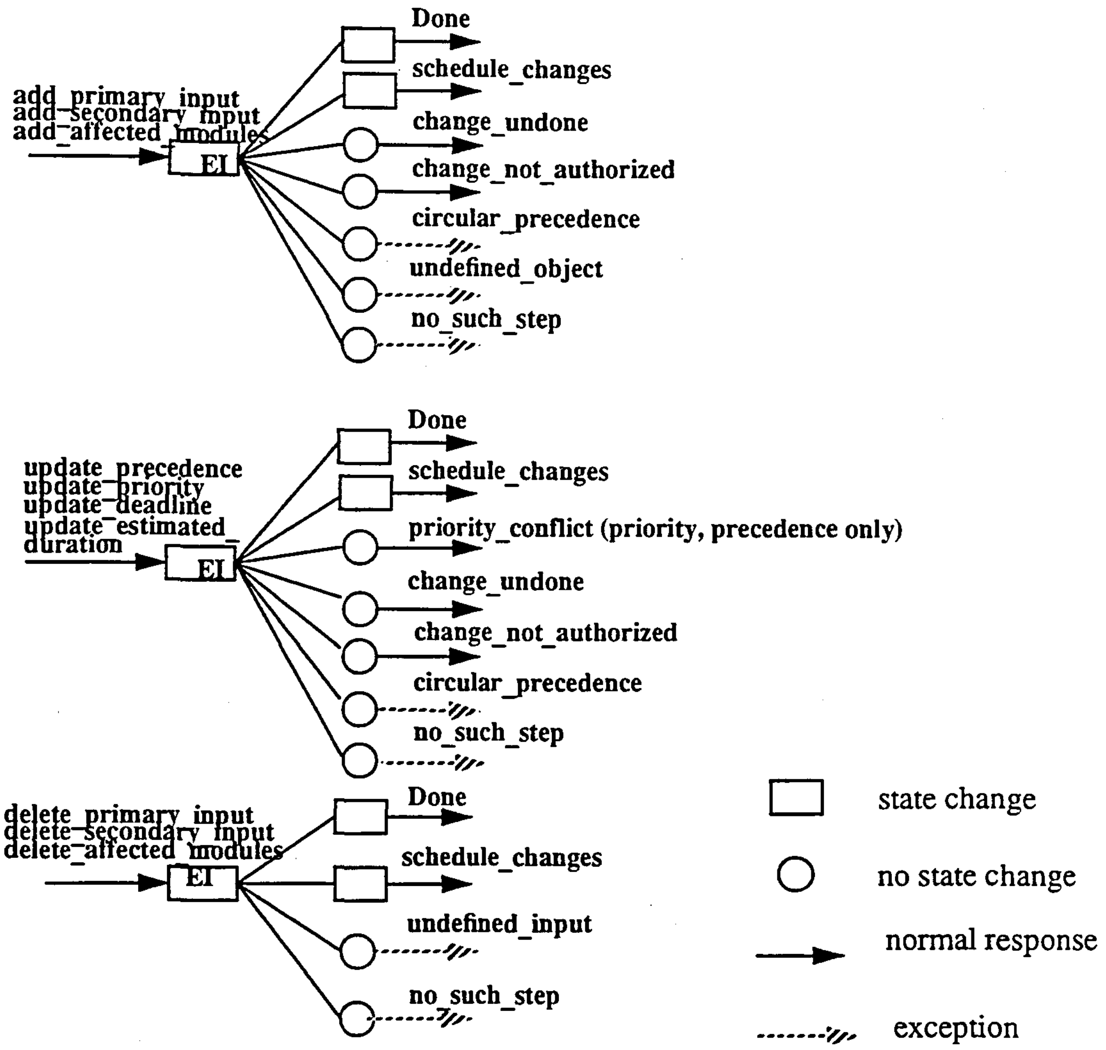
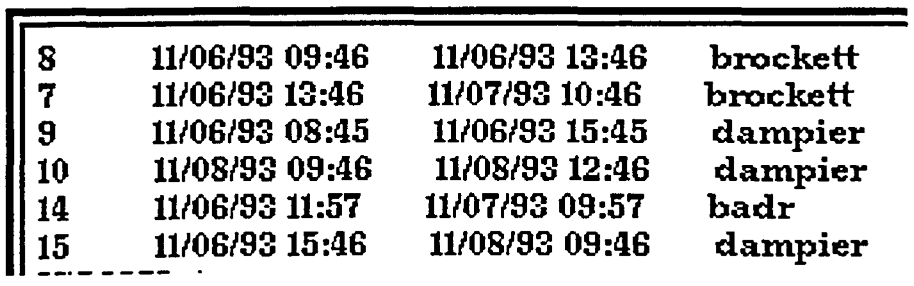


FIGURE 6. Stimulus Response diagram for the edit interface

The ECS automatically monitors changes in plan and takes the appropriate action to maintain the required constraints. The following scenario shows some of the ECS system features.

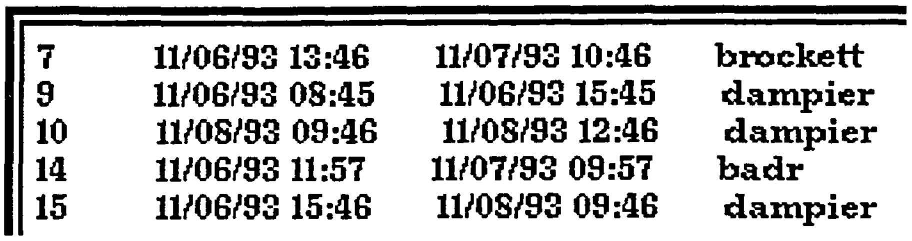
Using the command show_schedule we get the current schedule of the planned steps as shown in the following screen image.



```
8     11/06/93 09:46    11/06/93 13:46    brockett
7     11/06/93 13:46    11/07/93 10:46    brockett
9     11/06/93 08:45    11/06/93 15:45    dampier
10    11/08/93 09:46    11/08/93 12:46    dampier
14    11/06/93 11:57    11/07/93 09:57    badr
15    11/06/93 15:46    11/08/93 09:46    dampier
```

To show the automated VCCM capabilities of the ECS let us commit the substeps of step 1 (steps 6,7, and 8, step 6 is already committed, the composite steps do not appear in the schedule) then step 1.

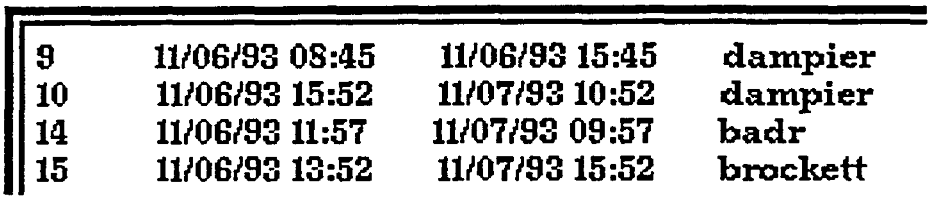
First let designer brockett commit step 8. This automatically updates the schedule as shown below. This leads to assigning brockett step 7 and sending him an e_mail message



```
7     11/06/93 13:46    11/07/93 10:46    brockett
9     11/06/93 08:45    11/06/93 15:45    dampier
10    11/08/93 09:46    11/08/93 12:46    dampier
14    11/06/93 11:57    11/07/93 09:57    badr
15    11/06/93 15:46    11/08/93 09:46    dampier
```

informing him about his new assignment.

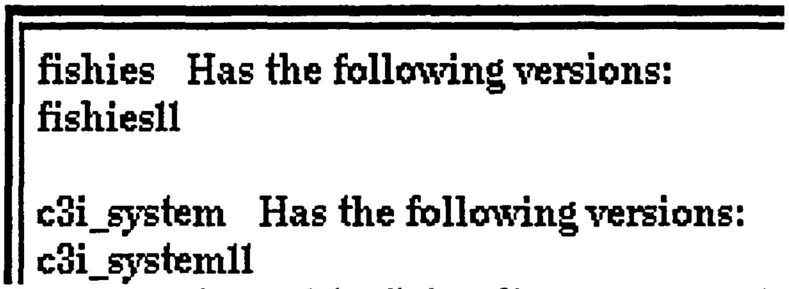
Now for the sake of the example let designer brockett commit step 7. This is an early commit which automatically updates the schedule as shown below.



```
9     11/06/93 08:45    11/06/93 15:45    dampier
10    11/06/93 15:52    11/07/93 10:52    dampier
14    11/06/93 11:57    11/07/93 09:57    badr
15    11/06/93 13:52    11/07/93 15:52    brockett
```

Notice that as soon as designer brockett commits step 7 the system assigns him step 15 which was planned for designer dampier before, because step 15 is ready and designer brockett becomes available after committing step 7.

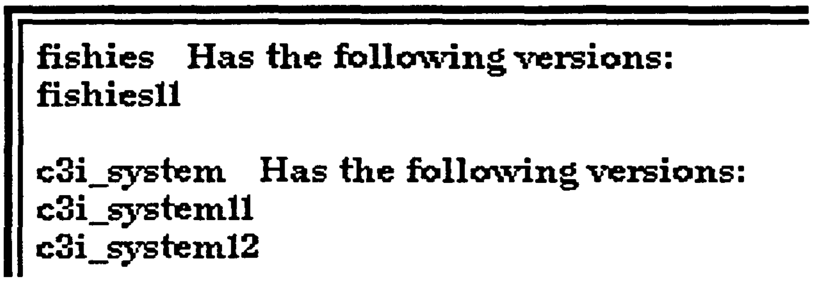
Before committing step 1 let us have a look at the versions of both c3i_system and fishies prototypes in the database using show prototypes command as shown below.



```
fishies   Has the following versions:
fishies11

c3i_system   Has the following versions:
c3i_system11
```

The manager commits step 1 (applied to c3i_system prototype) using commit step command from his menu when all the verification and checking for the substeps are done. The result of this command is creating version number 2 on variation number 1 of the c3i_sysem as shown below.



```
fishies   Has the following versions:
fishies11

c3i_system   Has the following versions:
c3i_system11
c3i_system12
```

Now if we look at the available steps at the system we notice that step 1 and its substeps 6, 7, and 8 all have the status completed when we use the show steps with the option completed from the manager menu as shown below.

```
step_set has 15 items.

6,        Status: completed
7,        Status: completed
1,        Status: completed
8,        Status: completed
```

One more feature of the ECS is related to the default base version to which the top step is applied. When step 1, 2, and 3 are created as top level steps they had the c3i_system 1:1 as the base version for the three steps. When step 1 is committed producing c3i_system 1:2 the default base version for both steps 2 and 3 is automatically changed to be the newly created version c3i_system 1:2.

Another important feature of the ECS is the automatic warning to both manager and designer one hour before a step is due to commit as shown in the E-mail message below received by the manager.

```
From badr Sat Nov  6 14:26:18 1993
Return-Path: <badr>
Received: from suns7-caps.cs.nps.navy.mil (suns7.cs.nps.navy.mil)
ps.navy.mil (4.1/SMI-4.1)
        id AA08946; Sat, 6 Nov 93 14:26:18 PST
Date: Sat, 6 Nov 93 14:26:18 PST
From: badr (salah badr)
Message-Id: <9311062226.AA08946@taurus.cs.nps.navy.mil>
To: badr
Status: R

ATTENTION REQUIRED  Step:  9        should commit within an hour...
```

a. Dropping a Designer

Designer dampier commits step 9, and the manager decides to schedule step 4 (step 4 has the substeps 11, 12, and 13). The updated schedule after committing step 9 is shown below.

```
10      11/06/93 15:52     11/07/93 10:52     dampier
14      11/06/93 11:57     11/07/93 09:57     badr
15      11/06/93 13:52     11/07/93 15:52     brockett
```

The manager uses schedule step command for step 4 then the ECS produces the updated schedule below.

Now the manager decided to send designer badr on a field trip, so he must delete him from the schedule. The manager uses drop designer option from the edit_team sub-menu. After the system asks for the manager's confirmation, it suggests deadline changes for both steps 13 and 12 as shown below.

```
10      11/06/93 15:52    11/07/93 10:52    dampier
14      11/06/93 11:57    11/07/93 09:57    badr
15      11/06/93 13:52    11/07/93 15:52    brockett
11      11/07/93 09:57    11/08/93 09:57    badr
13      11/08/93 09:57    11/08/93 13:57    dampier
12      11/08/93 09:57    11/08/93 14:57    badr
```

```
NOTICE:  The Designer just deleted was busy
         RESCHEDULING his/her tasks.
in-feasible schedule: step # 13
suggested deadline should be >= 20
Would you like to change it?  Answer(y/n)y

Enter the new Deadline 20
in-feasible schedule: step # 12
suggested deadline should be >= 21
Would you like to change it?  Answer(y/n)y

Enter the new Deadline 21
STEP_ID   S_LEVEL  D_NAME        START_TIME  FINISH_TIME
-------   -------  ----------    ----------  -----------
   14     LOW       dampier          3           9
   11     LOW      brockett          8          16
   12     LOW      brockett         16          21
   13     LOW       dampier         16          20
```

When the suggested deadline changes is accepted by the manager, the ECS produces the following updated schedule.

```
10      11/06/93 15:52    11/07/93 10:52    dampier
14      11/07/93 10:59    11/08/93 08:59    dampier
15      11/06/93 13:52    11/07/93 15:52    brockett
11      11/07/93 15:59    11/08/93 15:59    brockett
13      11/08/93 15:59    11/09/93 11:59    dampier
12      11/08/93 15:59    11/09/93 12:59    brockett
```

Notice that the assigned and the planned steps for designer badr are rescheduled to both designers brockett and dampier.

D. CONCLUSION

Integrating planning and version control allows both parts to be more fully automated. This technology may also enable a single manager to handle projects of larger size by providing decision support and taking care of low level details.

REFERENCES

[1] Badr Salah, "A Model and Algorithms for a Software Evolution Control System", Ph. D. Dissertation, Computer Science Department, Naval Postgraduate School, December 1993.

[2] Dampier D., Luqi, "A Model for Merging Software Prototypes", Technical Report, NPS CS-92-014.

[3] Feiler P. H., "Configuration Management Models in Commercial Environments", Technical Report CMU-91-TR-7, ESD-91-TR-7, 1991.

[4] Feldman S. I., "Software Configuration Management: Past Uses and Future Challenges" Proceedings of 3rd European Software Engineering Conference, ESEC '91, Milan, Italy, October 1991

[5] Hong K. and Leung J., "On-Line Scheduling of Real-Time Tasks" Real-Time Systems Workshop, May 1988.

[6] Luqi, "A Graph Model for Software Evolution", IEEE Transaction on Software Engineering. Vol. 16. NO. 8. Aug. 1990. pp. 917-927.

[7] Mostov I., Luqi, and Hefner K., "A Graph Model for Software Maintenance", Tech. Rep. NPS52-90-014, Computer Science Department, Naval Postgraduate School, Aug. 1989.

[8] Stankovic J. A., Ramamritham K., Shiah P., and Zhao W., "Real-Time Scheduling Algorithms for Multiprocessors", COINS Technical Report 89-47.

[9] Xu Jia., "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations", IEEE Transactions on Software Engineering, Vol. 19, No. 2, February 1993.

NOTES

DATE DUE			
1/9/97			
~~SEP 2 2~~	~~2015~~		
	JLC		