(1)

# ARO/AFOSR/ONR Workshop

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

**Software Slicing, Merging and Integration**

October 13 - 15, 1993

U.S. Naval Postgraduate School
Monterey, California

# Proceedings of the ARO/AFOSR/ONR Workshop

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

**Software Slicing, Merging and Integration**

**Sponsored by**

**Army Research Office**
**Air Force Office of Scientific Research**
**Office of Naval Research**
**Naval Postgraduate School**

**October 13 - 15, 1993**

**U.S. Naval Postgraduate School**
**Monterey, California**

**Workshop Chairman:**
**Valdis Berzins**

**Program Committee Chairs:**
**Luqi**
**Dave Dampier**

# List of Attendees

Hiralal Agrawal
Bellcore, MRE2D-388
445 South Street
Morristown, NJ 07960
(201) 829-5023
hira@bellcore.com

Sergio Antoy
Portland State University
Computer Science Department
Portland, OR 97207
(503)725-3009
antoy@cs.pdx.edu

Salah Badr
Computer Science Department
U.S. Naval Postgraduate School
Monterey, CA 93943
(408) 656-2903
badr@cs.nps.navy.mil

Valdis Berzins
Computer Science Department
U.S. Naval Postgraduate School
Monterey, CA 93943
(408) 656-2461
berzins@cs.nps.navy.mil

A. Berztiss
University of Pittsburgh
Computer Science Department
Room 321, Alumni Hall, University Drive
Pittsburgh, Pennsylvania 15260
(412) 624-8401
alpha@cs.pitt.edu

Jim Brockett
Computer Science Department
U.S. Naval Postgraduate School
Monterey, CA 93943
(408) 656-2468
brockett@cs.nps.navy.mil

Mauricio Cordeiro
Computer Science Department
U.S. Naval Postgraduate School
Monterey, CA 93943
(408) 656-2671
cordeiro@cs.nps.navy.mil

David A. Dampier
Computer Science Department
U.S. Naval Postgraduate School
Monterey, CA 93943
(408) 656-2912
dampier@cs.nps.navy.mil

Martin Feather
Suite 1000, 4676 Admiralty Way
Marina Del Rey, CA 90292-6695
(310) 822-1511
feather@isi.edu

Bill Griswold
Department of CS and Engineering
University of California San Diego
9500 Gilman Drive
La Jolla, CA 92093
(619) 534-6898
wgg@cs.ucsd.edu

David Hislop
U.S. Army Research Office Elec Div
4300 S. Miami Blvd
Research Triangle Park, NC 27709-2211
(919) 549-4255
hislop@aro-emh1.army.mil

J.C. Huang
Department of Computer Science
University of Houston
Houston, TX 77204-3475
(713) 743-3350/3366
jhuang@cs.uh.edu

Deepak Kapur
Department of Computer Science
State University of New York
Albany, New York 12222
(518) 442-4281
kapur@cs.albany.edu

Yuh-Jeng Lee
Computer Science Department
U.S. Naval Postgraduate School
Monterey, CA 93943
(408) 656-2361
ylee@cs.nps.navy.mil

Luqi
Computer Science Department
U.S. Naval Postgraduate School
Monterey, CA 93943
(408) 656-2461
luqi@cs.nps.navy.mil

A. Mili
Faculty of Science
University of Ottawa
150 Louis Pasteur/Priv.
Ottawa, Ontario K1N 6N5
Canada
(613) 564-9234
amili@csi.uottawa.ca

R. Mili
Kanata Software Engineering Services
75 Waterton Crescent
Kanata, Ontario K2M 1Z2
Canada
rmili@csi.uottawa.ca

Roland Mittermeir
University Klagenfurt, Inst F Informatik
Universitaetsstr 63, A-9022
Klagenfurt
Austria
(463) 531-7575
roland@ifi.uni-klu.ac.at

Frank Palazzo
Computer Science Department
U.S. Naval Postgraduate School
Monterey, CA 93943
(408) 656-2225
palazzo@cs.nps.navy.mil

John Salasin
ARPA/SISTO
3701 N.Fairtax Dr.
Arlington, Virginia 22203-1714
(703) 908-8207
jsalasin@sei.cmu.edu

Alan Shaw
Department of Computer Science
University of Washington
Seattle, Washington 98195
(206) 543-9298
shaw@cs.washington.edu

Mantak Shing
Computer Science Department
U.S. Naval Postgraduate School
Monterey, CA 93943
(408) 656-2634
mantak@cs.nps.navy.mil

Y.V. Srinivas
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304
(415)493-6871
srinivas@kestrel.edu

David Stemple
Department of Computer Science
University of Massachusetts at Amherst
Amherst, MA 01003
stemple@cs.umass.edu

Leon Sterling
Case Western Reserve University
Cleveland, OH
(216) 368-5278
leon@alpha.ces.cwru.edu

Douglas Waugh
SEI
801 N. Randolph St. Suite 405
Arlington, Virginia 22203
(703) 908-8207
dww@sei.cmu.edu

Ray Strong
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
(408) 927-1758
strong@almaden.ibm.com

Ed Wimmers
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
(408) 927-1882
wimmers@almaden.ibm.com

# Table of Contents

# Preface

**1993 ARO/AFOSR/ONR Workshop on**
**Increasing the Impact of Formal Methods for Computer-Aided Software Engineering**

**by**
**Valdis Berzins**
**Computer Science Department, U.S. Naval Postgraduate School, Monterey, California**

The U.S. spends billions of dollars per year on software, much of it for software modifications and maintenance. Computer aid should give software designers better control over their products, with resulting improvements in software usefulness and reliability and reductions in time and cost for large scale changes. Our basic premise is that appropriate formal methods supported by appropriate software tools can be very beneficial for practical software development. We believe that it is possible and necessary to validate this premise and to put it into common practice. As part of this effort, we would like to ask for your help to establish and support an important research direction, computer aided software evolution.

## 1. Computer Aided Software Evolution

Aspects of computer aided software evolution have been studied for many years but the field has not yet gained wide recognition and support, partially because the capabilities of existing theoretical solutions are completely dwarfed by the complex and demanding needs of real software development projects. Much early work in the area addressed relevant but simplified subproblems, ignoring significant parts of real practical problems. These partial solutions have not been integrated to solve the real problem because the long term goals in this area have not been not well understood by researchers and funding agencies.

This is not the case in other disciplines such as operations research, physics, and applied mathematics. The difference is that we have not educated ourselves or the funding agencies about the connections among the different narrow areas of research, and the relationships to pressing national problems that span many of those areas. Everyone is spread thin. There are many aspects to the subject of software evolution and it is not easy for any single individual to recognize all of the connections among the different aspects. We need a forum to bring us together and strengthen these connections. We are very happy that funding agencies and researchers are willing to listen, even if they are primarily concerned with (apparently) different areas.

For computer support to provide enough practical benefit to justify the necessary investment in tools and training, we need a better theoretical understanding of software modification and effective formal methods for solving various subproblems associated with software evolution. We also need an effective framework for integrating these methods and their associated tools, and we need to validate our theoretical models against real software development efforts to make sure we are addressing the right questions.

Problems and solution procedures must be well understood before they can be successfully automated. Much effort has been spent on building big software tools with nice interactive human interfaces but with very little real power inside the wrappings. This has led to an increasingly widespread perception that investment in computer tools for software development does not pay off. Such a practice is wasteful on a large scale, and the perception it creates is very dangerous because it will block all hope of real progress if it is left unchecked.

Our field of study should develop the theoretical understanding and algorithmic solutions that will enable us to put powerful engines inside the pretty interfaces. It is our responsibility to make the capabilities of these new engines match the most urgent practical needs in software development and evolution.

## 2. Workshop Goals and Procedures

The goals of this workshop are to:

(1) Identify the key technical problems in computer aided software evolution.

(2) Clarify the relations to other parts of Computer Science.

(3) Examine plausible technical approaches and assess advantages and disadvantages.

(4) Identify directions for future work that can have a practical impact.

To provide the background for the workshop, we briefly survey some previous work and indicate its relevance to computer aided software evolution. This introduction will be followed by presentations on different aspects of the subject, interleaved with discussions to bring out implicit assumptions, clarify relationships between different points of view, and make assessments. The conclusions of each session will be summarized by the reporter for the session, and the results will be integrated into a workshop report.

We have a great deal to learn from each other, and this workshop can only start the process. We should get together again next year to assess progress and to take the next steps towards practical application of formal methods in software development.

## 3. Survey of Technical Problems

Software evolution has both global and local problems. The global problems concern modifying software systems with many variants and coordinating concurrent changes to the same system. The local problems concern a single engineer working on a single change. This workshop is focused on the global problems to keep the scope manageable. We hope that the local problems can be addressed in a future workshop.

To support the process of combining (and recombining) efforts of different people, we are interested in formal methods and algorithms for checking whether a set of changes is compatible, for combining a set of compatible changes, and for reconciling incompatible changes. A related issue is coordinating a set of changes being developed concurrently by a team of engineers. This raises problems of preventing inconsistencies between concurrent changes or detecting and reconciling inconsistencies, based on the partial and uncertain information available while a change is in progress.

The basic assumption of change merging is that the behavior of a system can be separated into a set of independent parts that can be recombined. Two changes are compatible (can be consistently combined) if the parts of the system behavior affected by each change are independent. A change affects a part of system behavior if the original version of the part is not equivalent to the changed version of that part. Behavior can be considered at different levels: computation traces, functions computed by programs, requirements satisfied by program behavior, etc. Each of these levels is associated with different notions of part, independence and equivalence.

### 3.1. Software Slicing

A program slice is a subset of a program whose computation trace is independent of the rest of the program[1, 12]. A slice includes the parts of a program that can influence the part of its behavior visible from a particular point of view, such as the value of a set of variables or output streams. Slices can thus be "independent parts" supporting change merging at the level of computation traces.

Previous approaches to slicing have mostly been based on data flow analysis of single-thread imperative programs. Some current problems include increasing the resolution of representations and methods for computing slices, developing slicing methods for wider classes of programming languages, and developing analogs at the specification and requirements levels. More detailed semantic models of program slicing can have the advantage of better resolution and the disadvantages of longer running times and possible divergence because exact slicing is not a computable function.

## 3.2. Software Change Merging

Software change merging is the process of combining modifications to software system behavior. We need methods that guarantee semantically correct results in all cases where they do not report conflicts. Some of the problems in this area are finding better models of changes to software, safe merging methods with fewer spurious conflict reports, methods for automatically resolving conflicts, accommodating changes to data types, module interfaces, optimizing changes that introduce different algorithms, proving safety of change merging procedures, etc.

One approach to semantically based change merging for programs is based on slicing [8]. Slicing has the advantage of efficiency and the disadvantages of being unable to combine changes that can reach the same output or changes that use different algorithms to compute the same function.

Another approach to change merging is based on meaning functions [4]. Meaning functions are the functions computed by programs. The images of individual input values can be considered to be independent parts, leading to high resolution change merging methods. Meaning functions can merge changes to the same output if both changes can not take effect for the same input and the same initial state, and they can accommodate algorithm changes such as speedup transformations. Meaning functions introduce the possibility of recognizing some equivalences between programs denoting different execution sequences that compute the same function. Methods based on meaning functions can involve large amounts of computation and can fail to terminate if not suitably constrained, because exact solutions to these problems are also not computable.

Change merging has also been investigated for specifications and requirements [5], although this work has not yet been carried down to the code level. Responses to different stimuli can be considered to be different components of a system's behavior, and the response to each stimulus can be specified by postconditions. This raises large scale issues such as merging changes to the set of stimuli recognized by a system, provides a means for modeling possible dependencies between responses to distinct system inputs, and introduces a looser interpretation for the equivalence aspect of change merging. In the context of meaning functions, two outputs are considered equivalent only if they are equal, because there is no weaker criterion for equivalence that is safe. In the context of specifications that are not completely tight, two distinct output values can be equivalent if they both satisfy the same postcondition.

## 3.3. Representing Software Design Decisions

Improved techniques for representing and reasoning about software design decisions are important for supporting change merging as well as for providing intelligent assistance for software development. The plan calculus was introduced in the programmer's apprentice project to help the system analyze the programmer's design rationale and to fill in implied details of partial designs [11]. Similar approaches are relevant to advances in change merging because more accurate techniques depend on the relations between programs and meaning functions or specifications.

Improved change merging methods will have to process specifications or meaning functions to combine changes, and then to transform these changes back into code. Although the general problem of synthesizing code from specifications is not likely to be solved soon, the type of code synthesis required for change merging is easier because pieces of code realizing all the parts of each version are already available. The code synthesis required is to properly recombine these parts, possibly with some adaptations of details. This process must ensure that putting the parts together in new ways still results in valid design justifications, which will require some reasoning support. Although this is a big problem, the process does not have to be "creative". In particular, it can be aided by knowledge of common patterns of program design, such as those captured via the plan calculus and a library of cliches.

## 3.4. Transformations

Monotonic transformations are another path to change merging. Monotonic transformations produce results that are compatible with the starting point, but which may be further constrained [6]. Thus these transformations preserve meaning, and may add refinements[2]. Monotonic transformations are relevant to change merging because a series of such transformations is one way to represent the dependence of lower level information on higher level information.

Past work on transformations has considered changes with a desire to replay the derivation of an implementation after a change to some of the earlier decisions. This capability is relevant to the problem of turning a specification change into the corresponding program change, which is one aspect of change merging. Direct consideration of the change merging problem in the context of transformations may also be fruitful: given a base version and two enhanced versions of a transformational implementation, can we determine which parts of the derivations correspond to the enhancements, and automatically construct a derivation that incorporated both enhancements.

## 4. Conclusions

Change merging is an attractive context for research and development related to computer-aided software construction because

(1) it is easier than unrestricted code synthesis, and

(2) much of the effort in software development is spent on modifications.

We have sketched some opportunities in this area and hope that the workshop will clarify the problems, identify new relationships to existing work, and suggest additional directions for progress.

## 5. Overview of the Position Papers

The position papers for the workshop fall into five main areas: software maintenance and evolution, software specification methods, software merging, slicing, and restructuring, software verification, testing, and synthesis.

The papers on software evolution focus on computer assistance. Luqi and Goguen describe some directions for making progress in software development using formal methods. Mittermeir and Kienzl discuss the use of intra-object schemas to automate certain kinds of changes to object classes. Srinivas and Smith propose a model of evolution in which an aspect of a system is changed and then the change is automatically propagated by constructing a minimal set of other changes that restore consistency.

The papers on specifications discuss the construction and evolution of specifications as well as some of the processes that can be supported by specifications. Feather surveys some efforts at incremental development of specifications. Mili and Mili describe a lattice structure for specifications that can support checking whether specifications of different aspects of a system can be consistently combined and materializing the combination if one exists. Berztiss evaluates the applicability of formal methods to different kinds of software systems, and in a companion paper describes an approach for re-engineering organizations by systematically identifying activities that can be incrementally automated. Shaw explores the adequacy of communicating real-time state machines for specifying large real-time systems. Stemple proposes an approach for extracting specification information from operational prototypes via derivation and proof techniques.

The papers on slicing and merging examine different aspects of program slicing related to automated synthesis and analysis of programs. Agrawal examines how to determine which gotos to include in a program slice. Dampier and Berzins examine slicing and change merging for a prototyping language with concurrency and real-time constraints. Griswold examines the connection between slicing and program restructuring. Huang examines the application of path decomposition, program slicing, and symbolic execution to program understanding and simplification. Sterling describes a method for merging simple PROLOG programs to automatically construct a complex one.

The papers on software verification, testing and synthesis examine various ways to use formal approaches to achieve reliable software. Antoy and Hamlet describe a method for computing test inputs that force a program down a specified path using the narrowing technique for solving symbolic equations. Cleaveland examines the utility of finite-state approaches to software verification. Kapur presents some recent advances in automated reasoning technology and assesses the potential for application to software development. Salasin and Waugh propose annotation of software architectures with obligations to satisfy non-functional requirements as a way to assess whether or not those requirements will be met in a complex system. Smith describes the transformational development of a class of transportation scheduling algorithms.

1. H. Agrawal and J. Horgan, "Dynamic Program Slicing", *SIGPLAN Notices 25*, 6 (June 1990), 246-256.
2. V. Berzins, "On Merging Software Extensions", *Acta Informatica 23*, Fasc. 6 (Nov. 1986), 607-619.
3. V. Berzins, "Software Merge: Semantics of Combining Changes to Programs", Technical Report NPS 52-91-4, Computer Science Department, Naval Postgraduate School, 1990. Revised for ACM Transactions on Programming Languages and Systems.
4. V. Berzins, "Software Merge: Models and Methods", *International Journal on Systems Integration 1*, 2 (Aug. 1991), 121-141.
5. V. Berzins and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1991.
6. V. Berzins, Luqi and A. Yehudai, "Using Transformations in Specification-Based Prototyping", *IEEE Transactions on Software Engineering* 19, 5 (May 1993), 436-452.
7. D. Dampier, Luqi and V. Berzins, "Automated Merging of Software Prototypes", *Journal of Systems Integration*, to appear.
8. S. Horwitz, J. Prins and T. Reps, "Integrating Non-Interfering Versions of Programs", *Transactions Programming Languages and Systems 11*, 3 (July 1989), 345-387.
9. G. Ramalingam and T. Reps, "A Theory of Program Modifications", *Proceedings of the Colloquium on Combining Paradigms for Software Development*, vol. LNCS 494, Springer-Verlag, Apr. 1991, 137-152.
10. T. Reps, "Algebraic Properties of Program Integration", *Science of Computer Programming 17*, 1-3 (Dec. 1991), 139-215.
11. C. Rich and R. Waters, "Knowledge Intensive Software Tools", *IEEE Transactions on Knowledge and Data Engineering 4*, 5 (Oct. 1992), 424-430.
12. M. Weiser, "Program Slicing", *IEEE Transactions on Software Engineering SE-10*, 4 (July 1984), 352-357.

## Goal of the Workshop

**The goal of the workshop is to try to answer the following questions:**

- **Can formal methods be used in practice?**
- **Are there formal methods that can scale up for large problems? If so, which ones?**
- **How can we develop a comprehensive and consistent set of formal methods that can address all phases of software development?**
- **What are the most important difficulties with current research on formal methods? What are the most important research problems for the future?**
- **How can research on formal methods be improved?**
- **How can we increase the practical impact of formal methods?**

# Some Suggestions for Using Formal Methods in Software Development*

Luqi
Computer Science Department, Naval Postgraduate School, Monterey, CA 93943
Joseph A. Goguen†
Programming Research Group, Oxford University Computing Lab

## 1 Introduction

This paper describes an approach to producing reliable and useful software systems. Appropriate research goals are identified for improving software quality through taking account of social factors, through formalization, and through computer aid for software analysis. synthesis, and certification tasks at all phases of software development, from requirements to maintenance.

Getting requirements right is crucial for success in software development; if the system does not do what is really needed, then it will not be accepted by its users. In fact, this is very common for large complex systems. Recent research suggests that most of the cost of software development arises from errors in requirements, and that the most significant of those errors arise through social, political and cultural issues [J]. Software evolution (sometimes called maintenance) is another major concern in this context because errors are often introduced as a system is modified, and evolution typically accounts for more than half of a software system's total life cycle cost. See [K] for discussion of some statistics on errors in software development.

## 2 Assessing the State of the Art

High reliability is desired for safety-critical systems. It is practically impossible to produce error-free software systems that solve complex real problems by purely manual methods, because human error rates are too high. However, complete automation of software development and evolution is not feasible in the near future. Some realistic near term research goals in this situation include:

1. Developing methods and tools for requirements that accurately reflect stakeholders needs and the social context of the proposed system. Support for communication and learning is needed because developers and clients must pool their knowledge to determine what a cost-effective system should do. Techniques from the social sciences are needed because people often cannot accurately describe what they actually do or how their organizations really operate.

2. Formulating a consistent set of mathematical models for a set of subproblems covering the software development process. This is needed to integrate methods and tools for different aspects of software development.

*The research reported in this paper has been supported in part by US Army Research Office, the National Science Foundation, the British SERC, ESPRIT Working Group IS-CORE, British Telecommunications, and Fujitsu Laboratories Limited.

†Also at SRI International, Menlo Park CA 94025.

3. Developing and certifying the correctness of automatic synthesis methods for tractable subproblems. In cases where this is possible, this approach provides gains in both reliability and productivity.

4. Developing interactive synthesis methods that guarantee absence of errors for less tractable subproblems. This approach combines the benefits of human creativity with the accuracy computer tools through sound formal methods.

5. Improving analysis and certification methods for of detecting and diagnosing errors in subproblems that cannot be covered by error prevention techniques. For aspects of the process must remain manual, computer assistance in locating and removing errors and in certifying that no errors are left is needed.

Most past work on improving software reliability has followed the last approach, largely at the code level. This has provided some useful tools without needing a full understanding of all the problems involved; It is also the least desirable direction for the future because the search for errors is often very expensive and because it is difficult to predict how many iterations will be needed to eliminate all errors. Thus error detection work is most reasonable for those aspects of software development where error prevention techniques are not feasible.

Successful execution of test sets constructed by random sampling over a probability distribution can give lower bounds on the mean number of executions between failures if the actual input values correspond to the given probability distribution [F]. This kind of statistical reliability assurance is sufficient in cases where input distributions are predictable and non-zero failure rates can be tolerated.

For some specialized classes of programs, there are methods to construct a finite set of test cases whose successful execution can establish correctness of the program for all possible inputs [D, F]. This is not possible in the general case: testing can show the presence of software errors but cannot certify their absence.

Work on program verification has produced methods for constructing and mechanically checking mathematical proofs that given programs meet given specifications for all possible inputs. This technology is not yet mature enough for practical applications; particular weaknesses of current technology include the following:

1. Proving that a program satisfies a given specification is useless without some assurance that the specification is valid, i.e., that it accurately represents the needs of the users. Systematic methods for validation of specifications are not well developed, and social issues may be critical in this case [J].

2. Since it can take more effort to construct correct code than to prove that the code meets a specification [E,I], aid for constructing the code together with the correctness proof is desirable.

3. Current systems require considerable human assistance, and the mathematical skills required to use these systems are beyond the abilities of many practicing software developers.

4. Tool support and the range of applicability of particular methods need further development.

## 3 Future Opportunities

Error prevention is possible both in cases where a software development task can be completely automated, and in cases where an automated tool realizes all of the designer's decision's in constrained ways that preclude mistakes. Some examples are meaning-preserving software transformations, which prevent divergences between specifications and the code [B], and syntax-directed editors, which prevent the creation of programs that do not conform to the syntax of the programming language.

It is commonly believed that error prevention is more difficult than error detection, but this is not always the case. For example, checking whether an equational specification for an abstract data type is consistent and complete is an undecidable problem. Nevertheless, there exists an error prevention technique that guarantees that every specification generated according to the rules is complete and consistent. These rules are simple enough to be applied and checked by a text editor, and they are sufficiently loose to accommodate the styles of specification that normally occur in practice [A].

Software development deals with information of many different kinds, at different levels of abstraction. We summarize some of the types of software analysis, synthesis and certification problems that should be investigated in Figure 1.

## 4 Conclusion

Advances in software analysis, synthesis and certification are essential for realizing trusted software systems. Work in this area should be expanded beyond the traditional approach of testing code in a programming language and proving that programs satisfy formal specifications, to include computer support at *all phases of software development from* requirements analysis to system evolution. Some key areas for future research include:

1. Methods for validating requirements and specifications, such as prototyping and techniques for testing prototypes and specifications assist user perceptions.

2. Methods for constructing programs that guarantee correctness with respect to formal specifications, such as program synthesis by meaning-preserving transformations and the certification of application-specific program generation schemes.

3. Approaches for making formal methods easier to use, reducing the amount of manual effort required, and for reducing the amount of training and mathematical skill required for practitioners to apply these methods, by designing software tools that hide theoretical complexities behind simple interfaces.

4. Methods relevant to software evolution, such as change merging, monotonic transformations for modifying specifications and programs, and incremental versions of conventional software analysis, synthesis, and certification methods.

5. Software analysis techniques addressing properties of parallel, distributed, real-time, and knowledge-based systems should be explored as well as those for sequential systems.

6. Further work on program testing is needed, to expand the domains in which firm conclusions about satisfying specifications can be drawn from finite sets of test cases

| Level | Type of Analysis/Synthesis |
|---|---|
| Requirements | capture: social issues; use of video<br>traceability and consistency: hypermedia truth maintenance<br>model validation: prototyping and simulation<br>subgoal verification: prototyping |
| Specification | adequacy: prototyping, operational scenarios<br>consistency: type and domain checking<br>safety: proofs<br>validation: paraphrasing, views, simplification<br>error prevention: refinement transformations |
| Design | large grain issues: system composition<br>verification: proof of decomposition<br>liveness: deadlock and starvation checking<br>robustness: impact of degraded hardware<br>design for testing: control and observation<br>performance: complexity analysis<br>feasibility: satisfiability proofs<br>error prevention: cliches; assumption checking |
| Coding | synthesis: meaning-preserving transformations<br>performance: time and space analysis; benchmarking<br>liveness: proof of (clean) termination<br>real-time: analysis of scheduling methods<br>generic units: analysis of component families<br>error detection: complete test sets<br>error location: weakest preconditions |
| Evolution | requirements recapture: social and traceability issues<br>rebuilding: edit and execute system design<br>change impact: symbolic differences<br>restructuring: meaning-preserving transformations<br>error prevention: change merging |

Figure 1: Types of Software Analysis and Testing

constructed by definite and effective methods, and to systematically check assumptions about the operating environment on which the design of a system depends.

## 5 References

[A] S. Antoy, P. Forcheri and M. Molfino, "Specification-based Code Generation", in Proc. 23rd Hawaii International Conference on System Sciences, IEEE Computer Society, Jan. 1990, 165-173.

[B] F. Bauer, B. Moller, H. Partsch and P. Pepper, "Formal Program Construction by Transformations - Computer-Aided, Intuition-Guided Programming", IEEE Trans. on Software Eng. 15, 2 (Feb. 1989), 165-180.

[C] V. Berzins, "Software Merge: Models and Methods", International Journal on Systems Integration 1, 2 (Aug. 1991), 121-141.

[D] J. Bicevskis, J. Borozovs, U. Straujums, A. Zarins and E. Miller, Jr., "SMOTL - A System to Construct Samples for Data Processing Program Debugging", IEEE Trans. on Software Eng. SE-5. 1 (Jan. 1979), 60-66.

[E] D. Good, "Mechanical Proofs about Computer Programs", in Mathematical Logic and Programming Languages, Prentice-Hall, 1985.

[G] Luqi, "Software Evolution via Rapid Prototyping". IEEE Computer 22, 5 (May 1989). 13-25.

[H] A. Mili, W. Xiao-Yang and Y. Qing. "Specification Methodology: An Integrated Relational Approach", Software Practice and Experience 16, 11 (Nov. 1986), 1003-1030.

[I] D. Craigen, S. Gerhart, T. Ralston, An International Survey of Industrial Applications of Formal Methods, NISTGCR 93/626, National Institute of Standards and Technology, *Gaithersburg*, MD. 1993.

[J] J. Goguen, C. Linde, Techniques for Requirements Elicitation, Proceedings of Interenational Symposium on Requirements Engineering. pp. 152-164, San Diego, CA, Jan. 4-6, 1993.

[K] B. Blum, Some Very Famous Statistics. The Software Practitioner, Vo. 1, Number 2, March-April 1991.

[L] D. Dampier, Luqi and V. Berzins. "Automated Merging of Software Prototypes", Proceedings of the Fifth International Conference on Software Engineering and Knowledge Engineering, pp. 604-611, San Francisco, California June 16 - 18, 1993.

[M] S. Badr and Luqi, "A Version and Configuration Model for Software Evolution", Proceedings of the Fifth International Conference on Software Engineering and Knowledge Engineering, pp. 225-227, San Francisco, California. June 16-18, 1993.

# Intra-Object Schemas to Enhance the Evolution of Software-Objects

**Roland T. Mittermeir, Klaus Kienzl**
**Institut für Informatik**
**Universität Klagenfurt**
**AUSTRIA**

{mittermeir, klauki}@ifi.uni-klu.ac.at

## 1. PREMISES AND POSITION

Objects are - among other things - characterized by a clear separation between interface and implementation.

In this paper we argue for extending this duality between the external appearance and the internal mechanics by a third concept, the intra-object schema. It should be based on sound formal semantics and provide a level of indirection, useful in supporting the maintenance activities of objects.

The intra-object schema can be seen from the background of two perspectives:

1. provide a semantic roadmap to the various subcomponents of an object, be they subobjects of complex objects or be they components of lower granularity such a methods or state holders (instance variables).
2. provide a "service channel" into the object on which high level modification operations on aspects of the structure of the object can be performed.

## 2. FACETTES OF OBJECTS

To establish a basis for this approach, we destinguish in the definition of objects the definition of the *structure of the object* from the definition of the *contents of the object*. Neither of these concepts is yet on the level of implementation; both are definitory aspects.

To demonstrate what we are referring to with this distinction let's use an analogy: With a truck, the number of axes and the maximal payload it might carry are structural properties, so is the general kind of payload it might carry (liquid, parcels, containers, ...). The actual load of the truck, its actual volume, weight, and specific nature though are properties of its contents. Likewise, if we have an object implementing some data structure, the accessing discipline, the number of entries it might maximally support or the variety of values it might assume would be structural properties while the actual value it exhibits at a given point in time or the actual number of entries (and their values) it contains during a snapshot are contents related properties.

In classical object oriented software development, these two aspects are not separated and it is left to the programmer or designer (within the limits provided by the programming language in use) to provide methods for changing the contents and within certain limits methods for changing the structure (e.g. repaint the truck, change the quotient for computing the sales tax).

On the basis of our separation, we distinguish the following four classes of objects:

1. **state changes** - modifications of the contents of an object.
2. **structure changes** - modifications of structural properties such that the "container" itself changes without necessarily inducing a change in the "user-state" of the container object.
3. **mapping changes** - changes within a method such that the change is more involved than one which can be handled by changing some parameter (algorithmic consequences).
4. **overall change** - any change not covered by the three classes mentioned above. Changes of this sort would necessitate a regular, hands-on maintenance operation.

## 3. ELEMENTS OF AN INTRA-OBJECT SCHEMA

To support mechanized object evolution, an intra-object schema should contain at least the following kind of information:

- description of the signature of the object,
- definition of virtual methods,
- specification of specific constraints methods would assume concerning their attributes,
- specification of specific constraints concerning the usage fo methods,
- specification of inter-object constraints which are to hold between the component-objects of complex objects,
- definition of the actual state space used by implemented base methods.

The distinction between *virtual methods* and *implemented base methods* which has been referred to above is to be understood in a similar way as the distinction between implemented base relations (or ground facts) and computed views (or derived results).

## 4. BENEFITS OF THE INTRA-OBJECT SCHEMA

On the first glance, introducing intra-object schemata would allow a clear separation between methods supporting classical user operations (O- and V-operations on the user state space) and structural operations, which would be implemented as schema updates.

Depending on the complexity of the operation involved such schema updates can be fully or partially supported by executable code. For highly involved operations, the schema would provide just a roadmap which would provide orientation for the maintenance programmer.

The following list should give an idea about such update operations which can be supported by an intra-object schema and which are to be implemented as schema update operations (note: the most complex among them might extend into the implementation though).

- modifying the value of some constant (without consequences for the dimensionality of the state space of the object,
- extending (or shrinking) the value set of some argument type,
- extending (or shrinking) the value set of the type of some state variable,
- extending (or shrinking) the dimensionality of the state space (to the extent supported by its actual implementation),
- modifying the type of some (or all) dimensions in the state space to the extent not prohibited by specific constraints,
- introducing new virtual methods,
- introducing new virtual state dimensions (and the associated methods),
- removing virtual state dimensions and their directly associated methods,
- removing virtual methods.

## 5. MERITS AND OUTLOOK

The approach outlined above should provide the following merits for software maintenance:

- Consistent (semi-automatic) high level modification via schema modification operations (e.g. modification of the data-space as far as type info is concerned and provision of a semantically clean treatment of genericity via method constraints made explicit at a single location).

- Easy composition and recomposition of virtual functionality, hence it provides for an architecture supporting normalization considerations.

- Explict expression of integrity constraints. These integrity constraints might be expressed to hold between regular user operations or even between privileged maintainance operations. Even when they are not of such a quality that they can be automatically maintained, they would at least provide support for testing local modifications of methods. This becomes especially valuable, when these modifications have been applied manually.

- The concept is extendible to provide constraints for runtime-maintenance (support channels). To accomodate this notion, the intra-object schema has to be further extended (e.g. by inclusion of modification checkpoints).

# A Theoretical Basis for Software Evolution

Yellamraju V. Srinivas and Douglas R. Smith
Kestrel Institute, 3260 Hillview Avenue,
Palo Alto, CA 94304 ({srinivas,smith}@kestrel.edu)

## 1 Introduction

We describe an approach to software evolution that uses the same tools and techniques that have been successfully used in the transformational development of software (e.g., in KIDS [Smith 90]). We obtain an integrated view of software development and evolution by considering what is preserved and what is changed by each process. Software development is a sequence of transformations which preserve functionality but usually change some intensional property such as performance. Software evolution is a dual process in which "evolution" transformations change functionality but preserve properties such as well-formedness and internal consistency.

## 2 A Model of Evolution

We view evolution as the *transition from one consistent description* to another (see Figure 1). Each such transition can be decomposed into three phases: (1) start with a consistent description, (2) change some aspect of the description (possibly introducing inconsistency), (3) minimally change other parts of the description to re-establish consistency (change propagation). Here are some examples of artifacts, observations (invariant properties to be maintained) on these artifacts, and changes which affect these properties.

EXAMPLE 2.1. Artifact: a program. Observed property: well-formedness. Change: modification of the signature of a function, say, by adding a parameter. Propagation: change all references to the function. □

EXAMPLE 2.2. Artifact: a theory interpretation $I$ from theory $A$ to theory $B$. Observed property: (the assertion that) $I$ is a valid theory interpretation. Change: some modification to theory $A$. Propagation: change $B$ and/or $I$ to obtain a new interpretation. □

EXAMPLE 2.3. Artifact: graphical representation of a data structure. Observed property: (the assertion that) a tree-like picture on the screen is a representation of an abstract tree stored inside. Change: any modification of the abstract tree. Propagation: update the display to reflect the change. □

EXAMPLE 2.4. Artifact: a transportation schedule. Observed property: satisfaction of timing, trip separation, capacity, etc. constraints. Change: decrease in available resources. Propagation: reschedule movements which use unavailable resources. □

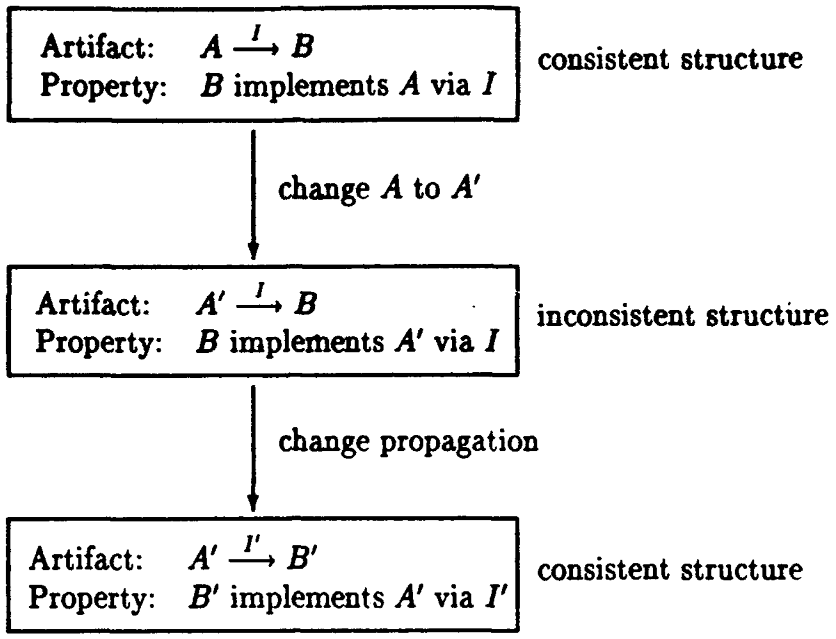

Artifact: $A \xrightarrow{I} B$
Property: $B$ implements $A$ via $I$
consistent structure
change $A$ to $A'$
Artifact: $A' \xrightarrow{I} B$
Property: $B$ implements $A'$ via $I$
inconsistent structure
change propagation
Artifact: $A' \xrightarrow{I'} B'$
Property: $B'$ implements $A'$ via $I'$
consistent structure

Figure 1: A model of evolution

Example 2.2 captures the essence of propagating changes in the requirements into an existing implementation. Example 2.3 characterizes situations where two or more interacting entities are mutually constrained.

## 3 Change Propagation

For change to be detectible, there has to be an observation. For, if an artifact is changed, and there is no change in the observed properties, then the change has no effect, and there is no need for change propagation. However, if the property we observe changes as a result of the change to the artifact, we have to do something to re-establish the observed property. This is the problem of change propagation.

Propagating arbitrary changes is a hard problem. Therefore, we consider a restricted class of changes, generalization and specialization, i.e., *monotonic* changes. Restricting the class of changes to monotonic changes represents the next step in expanding our current knowledge about change propagation. "Equational change," i.e., changing an entity into an equivalent entity has been well explored, e.g., in rewriting, transformation systems, and theorem-proving. Arbitrary change is too unconstrained, and ill-focussed. Monotonic change (i.e., generalization or specialization) seems structured and constrained enough, yet, surprisingly, encompasses a variety of situations (see examples above).

To model change, we associate a partial order (or several partial orders, if necessary) with the type of each entity which can vary. Each partial order represents monotonic changes along some dimension.

EXAMPLE 3.1. Typical examples of partial orders representing variation are the natural numbers with the less-than relation, trees with the subtree relation, sets with the subset relation, sequences with the subsequence relation, formulas with the implication relation, datatypes with the subtype relation, etc. Corresponding to the partial orders above, examples of the class of changes we consider are: strengthen or weaken a formula, expand or contract a set, replace a numeric expression by an upper or lower bound, etc. □

**Variance.** We analyze the dependence of the observed property (which we wish to maintain) using the notion of *variance*. Variance indicates the dependence of a term on its subterms: it is the direction and amount of change of a term with respect to changes in the subterms (variance is similar to the notion of polarity that is used in logic).

EXAMPLE 3.2. The length of a sequence and the size of a tree are purely covariant functions; the ordering on sequences is the subsequence relation, that on trees is the subtree relation.

$$\frac{x \text{ is a subsequence of } y}{\text{length}(x) \leq \text{length}(y)} \qquad \frac{s \text{ is a subtree of } t}{\text{size}(s) \leq \text{size}(t)} \qquad \frac{T \subseteq S}{(\forall x \in T \cdot \phi(x)) \Leftarrow (\forall x \in S \cdot \phi(x))}$$

Universal quantification is purely contravariant in the quantified set; the ordering on sets is the subset relation, that on formulas is given by implication: $\phi \leq \psi$ if and only if $\phi \Rightarrow \psi$. □

**Directed Inference.** Knowing the invariant formula to be maintained and the variance of its parts, we can determine the directions in which to change the parts of the formula to re-establish its truth. The change propagation is done by directed inference [Smith 82, Smith 90]: constrained by the direction of increasing the truth-value of the formula, we generalize/specialize parts of the formula (as determined by variance) until the formula becomes true. We recursively apply this procedure to change the parts, until we reach entities for which we have explicit operators to effect the required change.

## References

[Smith 82]

SMITH, D. R. Derived preconditions and their use in program synthesis,. In *CADE 6* (1982), *Lecture Notes in Computer Science,* Vol. 138, Springer-Verlag, pp. 172–193.

[Smith 90]

SMITH, D. R. KIDS: A semiautomatic program development system. *IEEE Trans. Softw. Eng. 16*, 9 (Sept. 1990), 1024–1043.

# Slicing Programs with Gotos

Hiralal Agrawal
Bellcore
*hira@bellcore.com*

## 1 Introduction

Program slicing has applications in many areas including program understanding, testing, debugging, maintenance, optimization, parallelization, and integration (see, e.g., [1, 3, 4, 5]). Algorithms to compute program slices are based on finding the closure of data and control dependences of relevant statements [4, 5]. Although algorithms to compute control dependences in the presence of goto statements have been developed [2], algorithms to decide which goto statements to include in a slice have obtained little attention. Even the "structured" derivatives of the goto statement such as the break, continue, and return statements, as in C, have not been adequately considered in this context. In this paper, we present an algorithm to determine which goto statements to include in a slice when the program being sliced contains goto statements. Unless otherwise specified, we will use the term goto to refer to both the goto statement as well as its structured derivatives mentioned above.

## 2 Finding the Relevant Gotos

A goto statement does not assign a value to any variable. Thus no statement may be data dependent on it. Also, as a goto statement is not a predicate, no statement may be control dependent on it. Therefore, the conventional slicing algorithm that finds the closure of data and control dependences of a given node may not cause any goto statements to be included in a slice.

It is easy to modify the conventional slicing algorithm to determine which conditional goto statements, such as those on lines 3 and 5 in Figure 1-a, to include in a slice: If the predicate in a conditional goto statement is included in a slice because some other statement is control dependent on it, then the associated goto must also be included. The predicate will not serve any purpose in the slice without the accompanying goto.

Figure 1-b shows the program slice of the program in Figure 1-a with respect to positives on line 15 obtained using the above scheme. Unfortunately, the slice, unlike the original program, fails to ensure that the assignment on line 8 is executed *iff* $x > 0$ as it does not include the relevant unconditional goto statements. Figure 1-c shows the correct slice with the relevant unconditional gotos included. Note that although the gotos on lines 7 and 13 are included, that on line 11 is not.

The same situation occurs in the presence of break, continue, and return statements which are special cases of the unconditional goto statement. The program in Figure 3-a shows a program with continue statements that is equivalent to the program in Figure 1-a. Figure 3-b shows the corresponding slice with respect to positives on line 17 obtained using the conventional slicing algorithm. Note that, without the relevant continue statements in the slice, the assignment on line 9 is incorrectly executed during each loop iteration irrespective of the value of x. Figure 3-c shows the correct slice with the relevant continue statements included. Note that although the continue statement on line 7 is included, that on line 12 is not.

The question, then, is: *how does one determine which unconditional gotos to include in a slice?* Consider a sequence, $S_1; S_2; S_3$, of three statements in a program. Suppose $S_1$ and $S_3$ belong to a slice obtained using the conventional slicing algorithm and $S_2$ does not. If $S_2$ is an assignment statement, then control always passes from $S_1$ to $S_2$ to $S_3$ in the original program. Deleting $S_2$ from the slice will cause control to transfer automatically from $S_1$ to $S_3$ in the slice. The same holds true if $S_2$ is a compound statement, e.g., an if or a while statement, and the body of the compound statement does not contain any explicit gotos.

On the other hand, if there are explicit goto statements in the body of $S_2$, then control need not always pass to $S_3$ after the execution of $S_1$ and $S_2$ in the original program, because the explicit gotos in $S_2$ may cause the control to transfer elsewhere. In this case we may not omit $S_2$ from the slice, as otherwise control will always pass unconditionally from $S_1$ to $S_3$ in the slice. We need not, however, include all statements in $S_2$ in the slice. We only need to include certain gotos in it along with their dependences. This is required because in the presence of gotos, the statement that lexically follows a statement in a program need not also be its immediate postdominator.

A statement, $S'$, is said to be the immediate *lexical successor* of a statement, $S$, in a program if deleting $S$ from the program will cause control to pass to $S'$ whenever it reaches the corresponding location in the new

```
1:       sum = 0;
2:       positives = 0;
3:  L3:  if (eof()) goto L14;
4:       read(x);
5:       if (x > 0) goto L8;
6:       sum = sum + f1(x);
7:       goto L13;
8:  L8:  positives = positives + 1;
9:       if (x%2 != 0) goto L12;
10:      sum = sum + f2(x);
11:      goto L13;
12: L12: sum = sum + f3(x);
13: L13: goto L3;
14: L14: write(sum);
15:      write(positives);
```

(a) a "goto" version of the example program

```
2:       positives = 0;
3:  L3:  if (eof()) goto L14;
4:       read(x);
5:       if (x > 0) goto L8;
8:  L8:  positives = positives + 1;
14: L14:
15:      write(positives);
```

(b) incorrect slice

```
2:       positives = 0;
3:  L3:  if (eof()) goto L14;
4:       read(x);
5:       if (x > 0) goto L8;
7:       goto L13;
8:  L8:  positives = positives + 1;
13: L13: goto L3;
14: L14:
15:      write(positives);
```

(c) correct slice

Figure 1: An example program with gotos, and its program slice with respect to positives on line 15

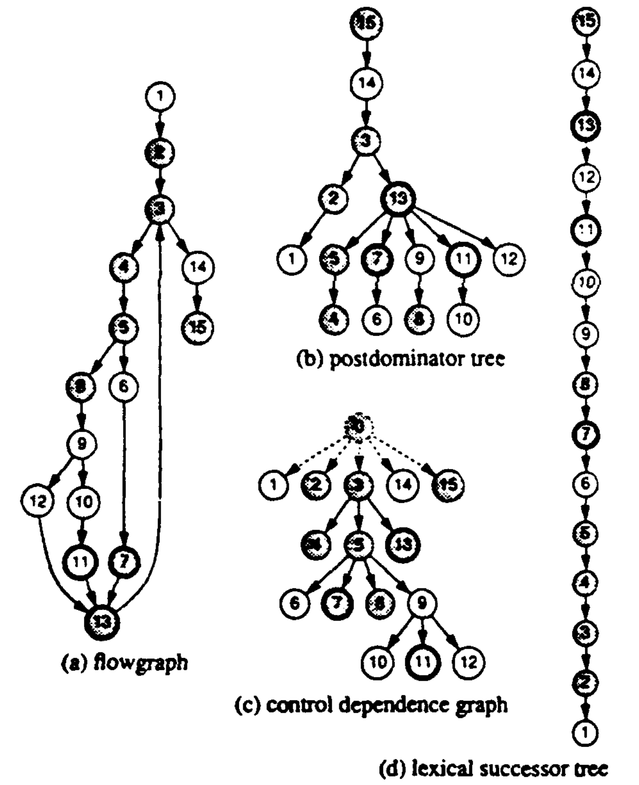


Figure 2: Various graphs of the program in Figure 1-a

```
1:  sum = 0;
2:  positives = 0;
3:  while (!eof()) {
4:      read(x);
5:      if (x <= 0) {
6:          sum = sum + f1(x);
7:          continue;  /* goto line 3 */
8:      }
9:      positives = positives + 1;
10:     if (x%2 == 0) {
11:         sum = sum + f2(x);
12:         continue;  /* goto line 3 */
13:     }
14:     sum = sum + f3(x);
15: }
16: write(sum);
17: write(positives);
```

(a) a "continue" version of the example program

```
2:  positives = 0;
3:  while (!eof()) {
4:      read(x);
5:      if (x <= 0) {
8:      }
9:      positives = positives + 1;
15: }
17: write(positives);
```

(b) incorrect slice

```
2:  positives = 0;
3:  while (!eof()) {
4:      read(x);
5:      if (x <= 0) {
7:          continue;  /* goto line 3 */
8:      }
9:      positives = positives + 1;
15: }
17: write(positives);
```

(c) correct slice

Figure 3: A "continue" version of the program in Figure 1-a and the corresponding program slice

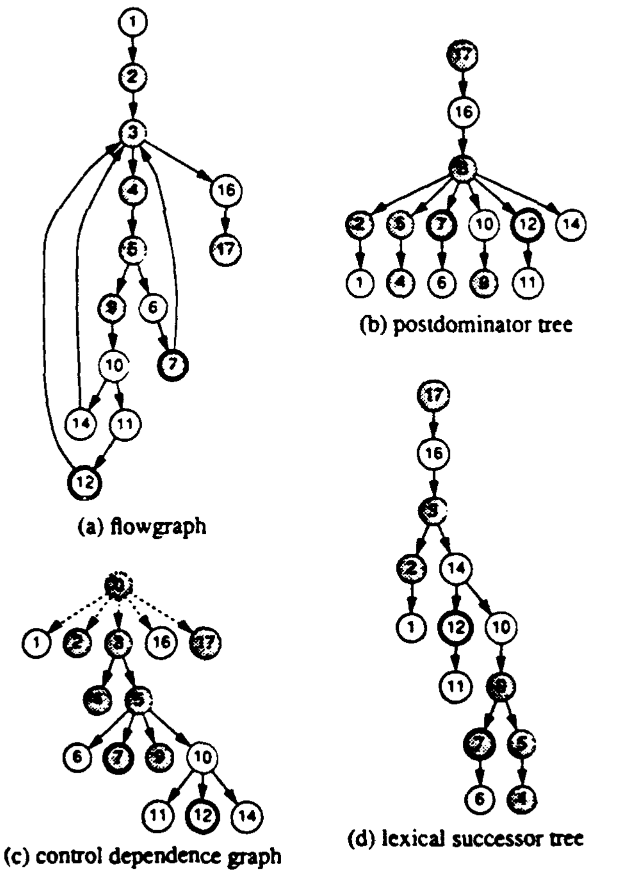


Figure 4: Various graphs of the program in Figure 3-a

program. Like the postdominator relationship, the lexical successor relationship may be represented graphically in the form of a *lexical successor tree*. It essentially depicts the nesting structure of the program and may be constructed in a purely syntax directed manner. Figures 2-d and 4-d show the lexical successor trees of the programs in Figures 1-a and 3-a, respectively.

In a "gotoless" program, the lexical successor of a statement is always the same as its immediate postdominator. Consequently, a slice of a "gotoless" program may be constructed by simply deleting statements that do not contribute towards the value of the variable in question. The statements that contribute towards the value of the variable in question are given by the conventional slicing algorithm. For a program that contains gotos, however, the lexical successor of a statement need not also be its immediate postdominator. Hence, we may not obtain a slice of such programs by simply deleting the statements that do not contribute towards the value of the variable in question. We must include other statements that ensure that the statements included in the slice by the conventional slicing algorithm get executed in the same relative order they are executed in the original program.

Suppose a statement, $S$, in a program is directly control dependent on a predicate, $P$. Let $D$ be the immediate postdominator of $P$. By definition of control dependence, we know that there exists a branch, $B$, out of $P$ such that whenever control takes this branch, $S$ is guaranteed to be reached before $D$ is reached. Suppose $P$ is included in a slice, *Slice*, obtained using the conventional algorithm but $S$ is not. If $S$ is an assignment statement, then we may omit it from the final slice without adversely affecting the control flow from $P$ to $D$ via $B$, because the lexical successor of an assignment statement is also its immediate postdominator. If, however, $S$ is a goto statement, we must include it in the final slice as omiting it from the slice may cause the control flow from $P$ to $D$ in the slice to deviate from the analogous control flow in the original program.

If $S$ is a predicate, then the decision about whether or not to include it in the final slice depends on whether or not its nearest postdominator in *Slice* is the same as its nearest lexical successor in *Slice*. If the two are different, we must include $S$ in the final slice, which in turn will cause the goto statements responsible for the difference to also be included in the slice. If, however, the nearest postdominator of $S$ in *Slice* is the same as its nearest lexical successor, then omitting $S$ from the slice will not adversely affect the control flow from $P$ to $D$. Figure 5 shows an algorithm to determine which statements to include in a slice when the program under consideration contains goto statements.

Figure 2 shows the flowgraph, the postdominator tree, the control dependence graph, and the lexical successor tree of the program in Figure 1-a. Node numbers in the graphs correspond to line numbers of the program statements. The nodes with thick outlines denote the unconditional goto statements. The control dependence graph also contains a dummy predicate node, viz., node 0. All top-level nodes—nodes that are not control dependent on any predicate in the program—are made control dependent on this node.

The shaded nodes in the graphs in Figure 2 indicate the statements included in the slice by steps 1 and 2 of the algorithm. The predicate on line 9 is the only remaining predicate not included in the slice during

```
1: Slice = the slice obtained using the conventional slicing algorithm;
2: Add any goto statements not in Slice, that are directly control dependent
   on predicates in Slice, to Slice;
3: Preds = the set of predicates not in Slice whose nearest postdominator in
   Slice is different from the nearest lexical successor in Slice;
4: while Preds is not empty do {
   4.1: Add all predicates in Preds, along with the closure of their data and
        control dependences, to Slice;
   4.2: Add any goto statements not in Slice, that are now directly control
        dependent on predicates in Slice, to Slice;
   4.3: Preds = the set of predicates not in Slice whose nearest postdominator
        in Slice is now different from the nearest lexical successor in Slice;
   }
5: return (Slice);
```

Figure 5: An algorithm to find program slices of programs with gotos. For programs that contain only "structured" gotos, the loop at step 4 is never entered.

these two steps. Figure 2-b shows that node 13 is nearest postdominator of node 9 in the slice. Figure 2-d shows that node 13 is also the nearest lexical successor of node 9 in the slice. Hence, in step 3 of the algorithm, *Preds* is evaluated to be the empty set and the loop at step 4 is never entered. Figure 1-c shows the resulting program slice.

Figure 6-a shows another version of the program in Figure 1-a where the indirect gotos from lines 7 and 11 via line 13 to line 3 have been replaced with direct gotos to line 3. Figure 7 shows the corresponding graphs for this program. As in the above example, the predicate on line 9 is the only predicate not included in the slice during the first two steps. The nearest postdominator of node 9 in the slice is node 3, as shown in Figure 7-b, whereas its nearest lexical successor in the slice is node 15, as shown in Figure 7-d. As the two are different, node 9 is included in the slice during step 4.1. This, in turn, causes step 4.2 to include the gotos on lines 11 and 13 in the slice. Figure 6-b shows the resulting program slice.

It can be shown that *for programs that contain only the "structured" gotos, if a predicate is not included in a slice by the conventional slicing algorithm, then its nearest postdominator in the slice is the same as its nearest lexical successor in the slice.* Thus, for such programs the loop at step 4 of the algorithm is never entered. Hence, for these programs we have a much simpler slicing algorithm. It is the same as the conventional algorithm with one additional step—*whenever a predicate is included in the slice, all gotos directly control dependent on it are also included in the slice.* Figure 3-d shows the corresponding slice of the program in Figure 3-a obtained using this algorithm.

Note that for programs that do not contain any gotos, no statement is added to the slice during step 2 of the algorithm. In this case the algorithm defaults, as desired, to the conventional slicing algorithm. Also

```
1:       sum = 0;
2:       positives = 0;
3:  L3:  if (eof()) goto L14;
4:       read(x);
5:       if (x > 0) goto L8;
6:       sum = sum + f1(x);
7:       goto L3;
8:  L8:  positives = positives + 1;
9:       if (x %2 != 0) goto L12;
10:      sum = sum + f2(x);
11:      goto L3;
12: L12: sum = sum + f3(x);
13:      goto L3;
14: L14: write(sum);
15:      write(positives);
```

(a) another "goto" version of the example program

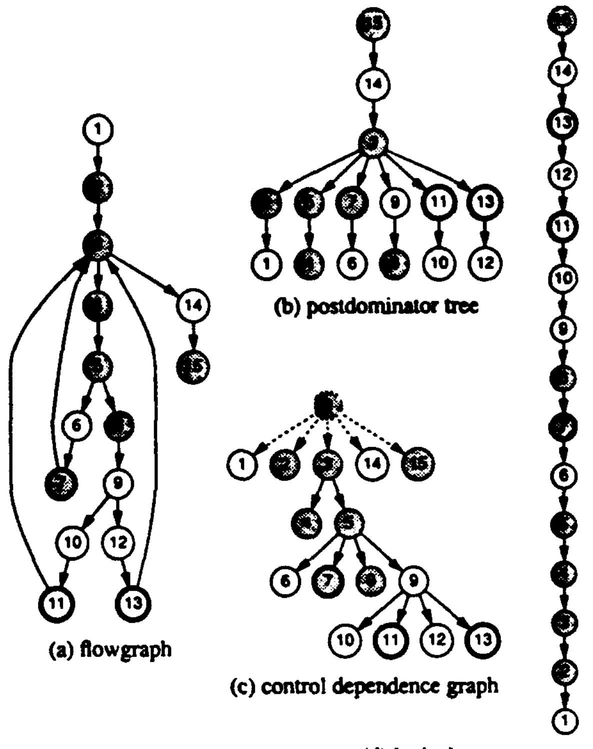
```
2:       positives = 0;
3:  L3:  if (eof()) goto L14;
4:       read(x);
5:       if (x > 0) goto L8;
7:       goto L1;
8:  L8:  positives = positives + 1;
9:       if (x %2 != 0) goto L12;
11:      goto L3;
12: L12:
13:      goto L3;
14: L14:
15:      write(positives);
```

(b) its correct slice

```
2:       positives = 0;
3:  L3:  if (eof()) goto L14;
4:       read(x);
5:       if (x > 0) goto L8;
7:       goto L1;
8:  L8:  positives = positives + 1;
9:       goto L3;
14: L14:
15:      write(positives);
```

(c) another slice

Figure 6: Another "goto" version of the program in Figure 1-a and the corresponding program slice



(d) lexical successor tree

Figure 7: Various graphs of the program in Figure 6-a

note that under this algorithm no special treatment of *conditional* goto statements is required. Whenever the predicate in a conditional goto is included in a slice, the associated goto is also included because the latter is directly control dependent on the former. Also, inclusion of the dummy node 0 in the control dependence graph ensures that all top-level goto statements are included in every slice, as the dummy node is included in every slice and all top-level gotos are directly control dependent on it.

A program slice is normally defined to be a *subprogram* of the original program. Some applications of program slicing, however, may not require that the slice necessarily be a subprogram of the original program. In this case, we may omit step 4 of the algorithm completely. Instead, we may simply insert a new statement, *goto D*, in place of each predicate, *P*, identified during step 3 of the algorithm where *D* is the nearest postdominator of *P* in the slice. Figure 6-c shows the resulting slice of the program in Figure 6-a obtained using this approach. Instead of including node 9 and the goto statements control dependent on it in the slice, a new statement, *goto L3*, is inserted at line 9 as node 3 is the nearest postdominator of node 9 in the slice.

## 3 Summary

For programs that contain goto statements, or even their structured derivatives, e.g., the break, continue, and return statements, the lexical successor of a statement is not necessarily the same as its immediate postdominator. For this reason, slices obtained using the conventional slicing algorithm of such programs may not preserve the behavior of the original program with respect to the slicing criterion. In this paper, we have proposed a new algorithm that alleviates this problem.

## References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.

[2] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[3] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[4] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[5] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

# A Slicing Method for Semantic Based Merging of Software Prototypes

David A. Dampier
Valdis Berzins
Department of Computer Science
Naval Postgraduate School, Monterey, CA 93943
e-mail: dampier@cs.nps.navy.mil, berzins@cs.nps.navy.mil

September 14; 1993

## 1 Introduction

Semantic based software merging is a problem which has been studied relatively little. Initial attempts at software merging were syntax based, and not very successful at preserving the semantic meaning of the programs [6, 7]. Later attempts to solve this problem have been focused on merging of extensions to programs [1] and integration of modifications to simple imperative programs [4]. We propose a method of merging prototypes based on slicing of enhanced data flow programs. We use the Prototyping System Description Language (PSDL) [5], which uses enhanced data flow diagrams to represent concurrent programs with real-time constraints.

Initial attempts at merging PSDL programs were syntax based and not completely successful [2]. Our current effort is centered on building a merged prototype with slices of the input versions which preserve the semantic meaning of those inputs. We present a method for finding slices of prototypes which affect particular data streams and a theorem which guarantees the behavior of the slice is the same regardless of whether it is contained in a larger prototype or is self-contained. Then, we introduce a method for change-merging prototypes based on this slicing definition.

## 2 Slicing of PSDL Prototypes

A *slice* of a PSDL prototype is defined in terms of the prototype's dependence graph [3]. It captures the portion of the prototype which affects the history of a set of streams. This is useful in isolating changes made to a base version of a prototype in a modification. If the slices of two versions with respect to the same set of streams are different, then there are significant changes that have been made to one version and not the other. A formal definition of a slice follows:

**Definition 1** Slice of a PSDL Prototype:

A *slice* of a PSDL prototype $P$ with respect to a set of data streams $X$, $S_P(X) = (V, E, C)$, is a subgraph of the PDG, $\mathbf{G}_P$, and is constructed as follows:

(1) $V$ is the smallest set that contains all vertices $o_i \in \mathbf{G}_P$ that satisfy at least one of the following conditions:

a) $o_i$ writes to one of the data streams in $X$.

b) $o_i$ precedes $o_j$ in $\mathbf{G}_P$, and $o_j \in V$.

(2) $E$ is the smallest set that contains all of the edges $x_k \in \mathbf{G}_P$ which satisfy at least one of the following conditions:

a) $x_k \in X$.

b) $x_k$ is directed to some $o_i \in V$.

(3) $C$ is the smallest set that contains all

of the timing and control constraints associated with each operator in $V$ and each data stream in $E$.

In order for a slice to be useful for merging, we must be confident that removing the slice from a larger prototype will not change its semantic meaning. Our prototype slicing theorem [3] demonstrates that this is the case.

Slicing Theorem for PSDL Prototypes:

Let $S_P(X)$ be the slice of a prototype $P$ with respect to a set of streams $X$. If $P$ and $S_P(X)$ have the same visible behavior on the input streams of $S_P(X)$, then:

$S_P(X)$ and $P$ have the same behavior on any subset of the streams in $S_P(X)$.

The proof of this theorem is a trans-finite induction over the length of the longest sequence of data vectors in the behavior of the slice. The basis for the proof is that since each stream contains an initial data value, and that value is part of the definition of the stream, then the initial data value for each stream is the same in the slice whether or not it is included in a larger prototype. The induction is on the length of the longest sequence of vectors in the behavior of the prototype. The induction step involves demonstrating through the use of the possibility function of each operator that we can construct the slice's behavior with length of no more than $k+1$, and that behavior is the same whether or not the slice is contained in a larger prototype or not. The last part involves demonstrating that the theorem is true for behaviors of infinite length, by showing that if a behavior of infinite length is different in or out of a larger prototype, then a finite prefix of that behavior must be different.

The significance of this theorem is that a slice captures a fragment of the semantic behavior of a prototype, and the behavior captured by that slice remains the same even if that slice is made a part of a different prototype, provided that it is also a slice with respect to that new prototype. This property is the basis for constructing a change merging operation that can provide semantic guarantees of correctness.

## 3 Slicing Method for Change-Merging

The method we propose for change-merging involves using prototype slicing to determine automatically which parts of the prototype have been affected by a change and which parts have been preserved. For example, consider the prototype in figure 1, and the changes to that prototype contained in figures 2 and 3.

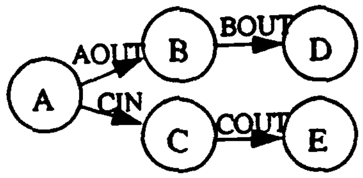


Figure 1: Base Version of a Prototype

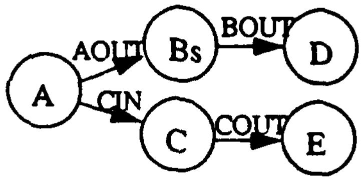


Figure 2: 1st Modification to the Prototype

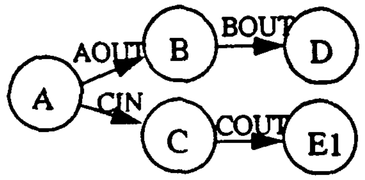


Figure 3: 2nd Modification to the Prototype

The merged version of this prototype, if feasible, must preserve the changes made in both of the modified versions. Using prototype slicing, we find the part of the base version, called the *preserved part* and shown in figure 4, and the parts of each modification which have different slices in the modified version and the base, called the *affected parts*and shown in figures 5 and 6.

These three subgraphs are then merged together to form the merged version shown in figure 7.

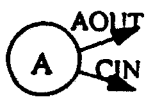


Figure 4: Preserved Part

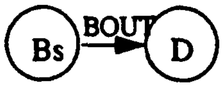


Figure 5: Affected Part of 1st Modification

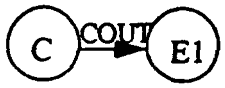


Figure 6: Affected Part of 2nd Modification

To ensure there are no conflicts between the two modifications, we check the slice of the merged version and each modification with respect to the streams in the affected part for that modification. If the slices are the same, then the semantic correctness of the merge is satisfied.

## 4 Conclusion

This method of change-merging for software prototypes shows great promise. An initial implementation of the change-merge tool has been completed and results of ongoing experimentation are positive. Possible future work for this project is making this tool recognize and automatically resolve some types of conflicts that currently have to be manually resolved by the designer.

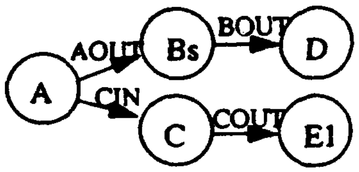


Figure 7: Merged Version of a Prototype

## References

[1] Berzins, V., "On Merging Software Extensions", **Acta Informatica**, Springer-Verlag, 1986, pp. 607-619.

[2] Dampier, D., **A Model for Merging Different Versions of a PSDL Program**, Master's Thesis, U.S. Naval Postgraduate School, Monterey, California, June 1990.

[3] Dampier, D., Luqi, and Berzins, V., "Automated Merging of Software Prototypes", **Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering**, KSI, June 1993, pp. 604-611.

[4] Horwitz, S., Prins, J., and Reps, T., "Integrating Non- Interfering Versions of Programs", **Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages**, Association for Computing Machinery, New York, New York, 13 - 15 January 1988.

[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] Siverberg, 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.

# An Architecture and Models for a Meaning-Preserving Program Restructuring Tool [*]

William G. Griswold
Department of Computer Science & Engineering, 0114
University of California, San Diego
San Diego, CA 92093-0114 USA
wgg@cs.ucsd.edu

## 1 Introduction

Tools that manipulate mulitple program representations are common. Optimizing and parallelizing compilers, program merging tools, and program restructuring tools are just a few examples. Constructing these tools is complicated, in part because it is hard to ensure that the diverse representations are kept consistent. Simplifying this task requires that models be provided to the tool builder for simplifying this task and that the realization be efficient enough for the tool user.

We approached this problem in the context of a program restructuring tool by developing a layered software architecture and developing—for certain key layers—models for how to build one layer from another. Although aspects of this approach are limited to semantically constrained applications like program restructuring, studying this approach may benefit the designers of related kinds of tools.

Automated assistance of program restructuring can overcome key aspects of program complexity [Griswold & Notkin 93]. A tool based on this technique uses a transformational approach by taking a locally-specified structural software change from the software engineer and performing additional compensating changes throughout the program to ensure that the meaning of the program does not change. If the tool cannot make meaning-preserving compensating changes, the engineer's change is disallowed and the problem is reported to assist the engineer in circumventing it.

Constructing a simple, efficient restructuring tool is hard because the tool uses data flow representations such as a Control Flow Graph (CFG) and a Program Dependence Graph (PDG) to compute the information needed to perform the semantic checks and compensating changes. The required interprocedural data flow analysis with alias (i.e., pointer) information is typically expensive to compute, and needs to be updated after each transformation applied by the tool user. A batch update approach is too slow, so instead the tool directly applies the equivalent of the syntactic change to the data flow representation [Griswold 93]. The direct update approach requires the translation of each Abstract Syntax Tree (AST) transformation procedure into an equivalent data flow representation transformation procedure. This translation task is performed by the transformation builder by examining the procedure implementing the AST transformation and deriving an equivalent procedure for the data flow representation. This process requires that the builder know the semantics of both representations and know the nature of the transformation being translated. The direct update approach can be modeled as the application of two equivalent functions to two different representations, one for the program text and one for the data flow information:

$$(F_{ast}(p_{old}), F_{pdg}(d_{old})) \longrightarrow (p_{new}, d_{new}).$$

To overcome the complexity of this approach, the system is layered to hide low-level concerns and AST updates are separated from data flow updates. Additionally, conceptual models assist in building key layers. These techniques apply to any direct update application, although the nature of direct update requires a semantically constrained application and an application-specific solution to translating the operations from the source representation to the target representation.

The principal layers for program restructuring were chosen as follows, from bottom to top: the basic representation layer for the AST and CFG/PDG, a layer on top for performing atomic updates on the AST and CFG/PDG, a layer for meaning-preserving updates on the AST and CFG/PDG (that is, $F_{ast}$ and $F_{pdg}$), and a layer combining the separate AST and CFG/PDG transformations (See Figure 1). Each layer builds on the layer below, adding one additional level of function. By supporting only one additional functional requirement per layer and keeping AST and CFG/PDG updates separate, the functions in each layer are smaller, and hence easier to understand and combine in the next layer. Providing the right abstractions in the higher layers eases the implementation of new transformations. Two models—described in the next section—assist in defining AST transformations and mapping them to an efficient CFG/PDG update.

[*]This work was supported in part by an IBM Graduate Fellowship, NSF Grants CCR-8858804 and CCR-9211002, a DEC External Research Program Grant, and the Xerox Corporation.

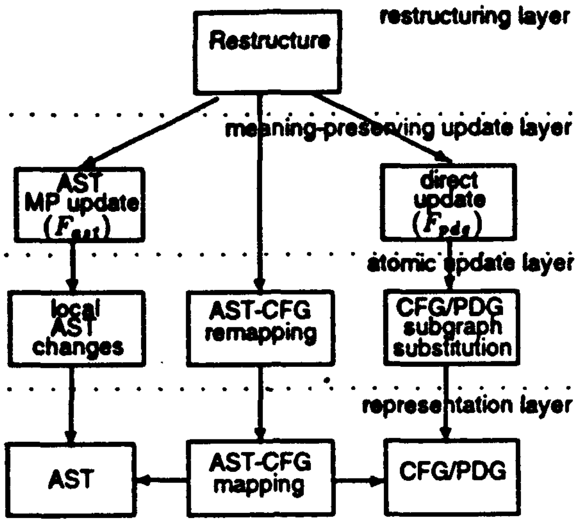


Figure 1: Layered Tool Design

## 2 Models for Designing Restructuring Transformations

Models of how to build one layer from another can help define an AST transformations and build an efficient CFG/PDG update for it. The globalization skeleton [Griswold 93] helps define a global meaning-preserving transformation from operations in the atomic update layer, based on the relationship between the tool user's initial change on a component and the tool's compensating changes to semantically affected components.

When invoking a transformation, the CFG/PDG needs to be updated to reflect the new AST structure. Although the transformation builder's knowledge of AST and CFG/PDG semantics must be applied in much of this process, some help is available by providing a small number of *PDG subgraph substitution rules* to help implement the CFG/PDG operations in the atomic update layer. These rules are in fact implemented as the atomic operations. To map an AST transformation (as described with a globalization procedure) to the PDG, the transformation builder selects a combination of PDG subgraph substitution rules that exactly describe the changes to the PDG that are needed to reflect the changes in the AST.[1] Then the implementations of those rules are used to construct a full meaning-preserving update on the CFG/PDG.

Although frequently two or more rules must be used to describe the required update, each rule alone describes a meaning-preserving change. Thus, in contrast to the globalization skeleton, which decomposes an AST transformation according to the syntactic separation of its changes, the PDG substitution rules divide the work according to structural and semantic responsibilities. For instance, there are different rules for changing data flow dependences and control dependences. When applied together to describe a single complete update, their substitutions change different aspects of the same subgraph. This approach to dividing a complex PDG update into smaller pieces eases understanding because it ensures the semantic integrity of each piece. The rules also specify preconditions for their correct application. These preconditions are the basis for the semantic checks performed before a tool transformation is applied.

Now one of the more common substitution rules can be introduced. In this rule only data flow and control dependences are shown. The handling of other edges is not difficult, but their inclusion here would obscure the algebra.

**The Distributivity Rule.** This rule is used, for example, to replace a variable reference with the expression that defines it, or vice versa. Intuitively, the rule states that if an expression's result is assigned to a variable, then a copy of that expression may replace a subsequent reference to that variable definition.

*Given the data flow dependence subgraph consisting of a vertex u with flow dependence successors $v_i$ carried by the edges $e_{s_i}$, and u' sequence-congruent[2] to u, then u' may acquire the flow dependence successor $e_s$ of u for any of the $v_i$ (Figure 2). The*

[1] These rules are described solely in terms of the PDG, rather than the combined CFG/PDG, because the substitutions on the CFG portion are straightforward due of its structural relationships to the AST and PDG.

[2] Rules like the distributivity rule replace a vertex or edge by an equivalent one to achieve a structural change. The Sequence-Congruence algorithm [Yang 90] provides a conservative definition of an equivalent vertex or edge. It computes equivalence classes of programs or subprograms that over the course of a program execution produce the same sequence of visible states. The Sequence-Congruence algorithm determines the equivalence of two PDG components

*label on the edge must be changed to $x$, a unique variable. Furthermore, there must not exist an edge $e'_s$ from a vertex other than $u$ to the affected vertex $v$*

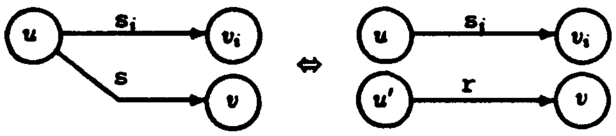


Figure 2: The Distributive Rule

This last qualifier disallows a second assignment to $s$ from flowing to $v$. When implementing the distributive rule, this restriction must be implemented as a runtime check that will prohibit the transformation if it fails. Multiple edges can be moved from $u$ to $u'$ by applying the rule to other $v_i$. The rule naturally extends to a cluster of vertices (i.e., to $u$ and its flow predecessors) by transitively applying the rule backwards through the graph. The vertex-equality part of the rule can be applied constructively by copying $u$ and its incoming edges, creating a trivially sequence-congruent $u'$. However, because replicating a vertex is essentially multiple evaluation of the vertex, it could redefine the output variable. This is the reason why the rule changes the edge variable label from $s$ to $r$.

Two other common rules are the transitivity rule and the control rule. The transitivity rule describes how chains of assignments between variables can be inserted or deleted. The control rule allows moving a definition of a variable to be moved under a conditional arm if all its uses are under that conditional arm.

## 3 Conclusion

The layering approach with the accompanying models supported successfully implementing a prototype restructuring tool. For restructuring, the globalization skeleton helps the transformation builder identify the changes to be performed and the program components involved; the PDG substitution rules help map the changes onto the PDG and CFG. The result is a systematic approach to implementing efficient direct updates of data flow representations for program restructuring.

Preliminary empirical comparisons indicate that direct updates are roughly constant time and fast enough to be used interactively, even in the prototype. Although these results could not be compared directly to other incremental techniques, the batch techniques used in the tool exhibit quadratic times with a large constant. When alias analysis in the batch algorithm is turned off (just for performance comparison), the times are closer to linear with a smaller constant, but still too slow to be used interactively.

One key question is whether there is a more formal method for deriving direct update transformations from source transformations. Denotational, operational, and data flow semantics techniques are promising [Cartwright & Felleisen 89][Ramalingam & Reps 89][Venkatesh 91][Parsons 92], but still in their early stages of development. Another question is whether there are PDG substitution rules for applications other than meaning-preserving restructuring.

## References

[Cartwright & Felleisen 89] R. Cartwright and M. Felleisen. The semantics of program dependence. In *Proceedings of the SIGPLAN '89 Conference on Programming Languages Design and Implementation*, July 1989. SIGPLAN Notices 24(7).

[Griswold & Notkin 93] W. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, July 1993.

[Griswold 93] W. G. Griswold. Direct update of dataflow representations for a meaning-preserving program restructuring tool. In *ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, December 1993. (To Appear).

[Parsons 92] R. Parsons. *Semantic Program Dependence Graphs*. PhD dissertation, Rice University, Dept. of Computer Science, April 1992. Technical Report No. Rice COMP TR93-202.

[Ramalingam & Reps 89] G. Ramalingam and T. Reps. Semantics of program representation graphs. Technical Report 900, Computer Sciences Department, University of Wisconsin, Madison WI, December 1989.

[Venkatesh 91] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Languages Design and Implementation*, June 1991. SIGPLAN Notices 26(6).

[Yang 90] W. Yang. *A New Algorithm for Semantics-Based Program Integration*. PhD dissertation, University of Wisconsin, August 1990. Computer Sciences Technical Report No. 962.

based on three properties, (1) the equivalence of their operators, (2) the equivalence of their inputs, and (3) the equivalence of the predicates controlling their evaluation. By definition, subgraphs of a PDG may be modified by the substitution of sequence-congruent vertices without changing the PDG's meaning.

# A Methodology for Program Understanding

*J. C. Huang*
*Department of Computer Science*
*University of Houston*
*Houston, TX 77204-3475*
*jhuang@cs.uh.edu*

The importance of being able to understand a program cannot be overstated. Given a program, whether the task at hand is to debug, to validate, to modify, or to reuse it, one needs to understand precisely what it does first. Although in theory this understanding can be achieved by reading the documentation, in practice the desired knowledge ultimately has to be gained or verified firsthand by studying the source code: it is the fact of life that software documentation is more than often inaccurate, incomplete, outdated, or nonexistent.

The methods of pathwise decomposition [HUAN90], program slicing [WEIS84], and symbolic execution [KING76] together provide us with a methodology for program understanding.

A program may become difficult to understand for many reasons. If it is so because of its large size or complex logical structure, one may overcome the difficulty by using the divide-and-conquer strategy, i.e., by understanding the program piecemeal.

Let P be a program written to implement a function f. In the process of designing and implementing P, the programmer decomposes f repeatedly until the algorithm used to compute f can be directly expressed in terms of the programming language used. Basically, there are three ways to decompose a function. The first is to decompose f into two functions $f_1$ and $f_2$ such that $f(x) = f_1(f_2(x))$. In a procedural language this is to be implemented in the form:

```
    a := f2(x);
    b := f1(a);
```

The second is to decompose f into three functions $f_1$, $f_2$, and $f_3$ such that $f(x, y, z) = f_1(f_2(x, y), f_3(y, z))$. In a procedural language this is to be implemented as

```
    a := f2(x, y);
    b := f3(y, z);
    y := f1(a, b);
```

The third way is to decompose f into two functions $f_1$ and $f_2$ such that

$$\begin{aligned} f(x) &= f_1(x) \text{ if } x \in D_1 \\ &= f_2(x) \text{ if } x \in D_2 \end{aligned} \qquad \text{or, alternatively,} \qquad \begin{aligned} f(x) &= f_1(x) \text{ if } P_1 \\ &= f_2(x) \text{ if } P_2 \end{aligned}$$

if the subdomains can be defined as $D_i = \{x \in D \mid P_i(x)\}$ for i =1, 2. In that case, the function f will be implemented in the form:

```
if $P_1$ then
        a := $f_1(x)$
else if $P_2$ then
        a := $f_2(x)$
```

The first step towards understanding a program piecemeal, therefore, is to identify the code segments that implement these subfunctions. These code segments are generally easier to understand because, in comparison with the whole program, they have a smaller size and simpler logical structure.

In practice, identifying a code segment that implements a subfunction is not as simple as it may appear in the above discussion. Two consecutive assignment statements or function calls in a program may not embody two subfunctions of a function. The use of loop constructs in a program further complicates definition and representation of subprograms.

The three methods mentioned previously can be used to identify (and possibly to simplify) the code segments discussed above.

To symbolically execute a program [KING76] is to identify the code segments in the program that embody the first kind of decomposition (and to recompose the function in the process). Thus a symbolic execution of

```
a := $f_1$;
b := $f_2$;
```

will yield $a = f_1$ and $b = f_2f_1 = f$ if $f_1$ and $f_2$ are subfunctions of f, and will yield $a = f_1$ and $b = f_2$ otherwise.

To slice a program [WEIS84] is to identify the code segments in the program that embody the second kind of decomposition. A slice of

```
1       a := $f_2(x, y)$;
2       b := $f_3(y, z)$;
3       y := $f_1(a, b)$;
```

with the slicing criterion (statement: 3, variable: a) is

```
a := $f_2(x, y)$;
```

and a slice of the critcrion (statement: 3, variable b) is

```
b := $f_3(y, z)$;
```

The method of pathwise decomposition [HUAN90] can be used to identify the code segments that embody the third kind of decomposition. A pathwise decomposition of

```
if P then
        a := $f_1(x)$;
else
        a := $f_2(x)$;
```

may yield a subprogram

```
/\ P;
if P then
        a := $f_1(x)$;
else
        a := $f_2(x)$;
```

which can be simplified to: /\ P; a := $f_1(x)$. Here /\ P is a *state constraint* [HUAN90], a shorthand notation of the following sentence:

*The program state at this point must satisfy predicate P, or else the program becomes undefined.*

This methodology can be used to identify all three types of functionally significant code segments in a program, and thus allows us to understand a program piecemeal. This methodology not only integrates several concepts in program analysis but also provides a seamless connection with the technique of proving program correctness by inductive assertions. To prove that a program is (partially) correct is to post-constrain the program with its output assertion, and then show that that constraint is tautological as explained in [HUAN90].

## References

HUAN90 J. C. Huang "State Constraints and Pathwise Decomposition of Programs," *IEEE Trans. on Software Engineering*, vol. 16, no. 8, Aug. 1990, pp. 880-896.

KING76 J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, July 1976, pp. 385-394.

WEIS84 M. Weiser, "Program Slicing," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 4, July 1984, pp. 352-357.

# On Merging Prolog Programs

*Position Paper*

**Leon Sterling**
Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio, 44106, USA.

email: leon@ces.cwru.edu

## Introduction

A critical issue for computer science is mastering the complexity of large software systems. There is a real danger of programmers drowning in a sea of details. Complexity threatens all aspects of the software lifecycle, from specification to coding to maintenance. One way to handle complexity is to find ways of building complex pieces from simple, well-understood pieces using automatic computer-aided methods.

This position paper advocates an approach to building complex Prolog programs from simple pieces. At its heart is a method for merging Prolog programs called *composition*. The approach has been successfully applied to the development of an explanation shell for Prolog-based expert systems, a trace facility, and a software testing program for applying coverage testing measures to programs written in C and Pascal.

Composition works on two programs that are related as sketched below. Indeed the relationship between the program drives composition. I do not believe it makes sense to merge two arbitrary unrelated programs.

We are able to prove properties of the merged program built by composition from the properties of the individual components. Being able to prove correctness results of composite programs is essential for handling complexity.

The mthod of composition has been developed for Prolog progrms but is applicable to functional languages, and even imperative languages. The key is in identifying structures which capture program behavior analogous to enhancement structures sketched below. The reason that composition emerged for Prolog is the nature of logical variables and the concise but clear programs that emerge naturally from writing in good style.

## Skeletons, techniques and enhancement structures

Our view of program merging comes directly from our underlying program development methodology. Prolog programs are best built systematically from two classes of standard components. *Skeletons* are (usually) simple Prolog programs with a well-understood control flow. Good examples are traversals of recursive data structures. Other examples are parsers and language interpreters. More discussion can be found in [3] and [4].

*Techniques* are standard Prolog programming practices. For example, the appropriate method for counting the number of elements of a list, the number of nodes in a tree, or the number of goal reductions or depth in a proof tree, is to increment an argument and then carry it as a context. Applying a technique to a program creates an *enhancement* of that program.

A relationship exists between an enhancement of a program, $P$ say, and the program itself, $Q$ say. Each clause $C_P$ in $P$ corresponds to a clause $C_Q$ in $Q$. For the basic merging

```
Input: A skeleton $S$
        Programs $P$ and $Q$ which have been derived from $S$ by application of techniques*, and
        are therefore enhancements of $S$.
Output: A program $P\_Q$ which combines the techniques of $P$ and $Q$ and which is an enhance-
        ment of $P$, $Q$ and $S$.
for each clause $C_S$ in $S$ do:
    find the corresponding clause, $C_P$, in $P$;
    find the corresponding clause, $C_Q$, in $Q$;
    construct a new clause $C_{P\_Q}$ such that the principal functor of the head of $C_{P\_Q}$ is the
    concatenation of the principal functors of the heads of $C_P$ and $C_Q$, and the head of
    $C_{P\_Q}$ contains the arguments of the head of $C_S$ followed by the additional arguments
    of the head of $C_P$ followed by the additional arguments of the head of $C_Q$;
    for each goal $L_S$ in the body of $C_S$ do:
        find the corresponding goal, $L_P$, in $C_P$;
        add the goals preceding $L_P$ which do not correspond to any goal in the body
        of $C_S$, and which have not been added already to the body of the new clause,
        $C_{P\_Q}$;
        find the corresponding goal, $L_Q$, in $C_Q$;
        add the goals preceding $L_Q$ which do not correspond to any goal in the body
        of $C_S$, and which have not been added already to the body of the new clause,
        $C_{P\_Q}$;
        construct a new goal whose principal functor is the concatenation of the
        principal functors of the goals $L_P$ and $L_Q$, and which contains the arguments
        of $L_S$ followed by the additional arguments of $L_P$ followed by the additional
        arguments of $L_Q$, and add the new goal to the body of the new clause, $C_{P\_Q}$;
    add any remaining goals in $C_P$ which do not correspond to any goal in the body of $C_S$,
    and which have not been added already to the body of the new clause, $C_{P\_Q}$;
    add the remaining goals in $C_Q$ which do not correspond to any goal in the body of $C_S$,
    and which have not been added already to the body of the new clause, $C_{P\_Q}$;
```

Figure 1: An algorithm for composition

algorithm a 1-1 correspondence between the clauses in $P$ and clauses in $Q$ is assumed, though we are also exploring more generality. For each literal $L_Q$ in $C_Q$ there is a corresponding literal $L_P$ in $C_P$. In general there will be additional goals in $C_P$. $L_P$ may have a different principal functor than $L_Q$, but it contains all the arguments of $L_Q$, possibly in different order, and may contain additional arguments.

We have developed a theory of program maps to formalize our notion of correspondence. The theory is described in [2]. In this position paper all concepts are presented informally.

Complicated programs are built by choosing a skeleton and repeatedly applying techniques. Separate enhancements of the same skeleton can be merged into a single program using a method we called composition. Figure 1 gives an algorithm for composition. The composition algorithm in Figure 1 merges two programs, but the algorithm can be repeated to merge more than two programs. A prototype tool has been implemented to support building Prolog programs from skeletons and techniques, and can automatically merge programs.

The sequence of operations used to build a complicated program can be represented graphically. An *enhancement structure* is a directed acyclic graph. Programs make up the nodes of the graph. Edges of the graph correspond to programming techniques. If a node A has out degree greater than one, then the program for A has been built by composition.

## Maintenance of Composed Programs

Programming techniques and their composition can form the basis of a calculus of operations on programs. Maintenance of programs can be accomplished by manipulating the techniques, a task far simpler than writing Prolog programs from scratch. As a result, program maintenance can be performed by non-specialist Prolog programmers, for which have growing anecdotal evidence. Indeed eventually the maintenance operations ideally will be performed in a graphical environment manipulating the enhancement structure of the program rather than the (more complicated) program text.

Enhancement structures can be used to analyze properties of merged programs. Proofs of correctness of the final programs can be built from the correctness of individual programs, which depend on the correctness of the composition algorithm. This work is described further in [1].

## References

[1] Marc Kirschenbaum, Ashish Jain, and Leon Sterling. Relative correctness of Prolog programs. Technical Report CES-93-19, Department of Computer Engineering and Science, Case Western Reserve University, 1993. Submitted for publication.

[2] Marc Kirschenbaum, Leon Sterling, and Ashish Jain. Relating logic program via program maps. *Annals of Mathematics and Artificial Intelligence*, 8(III-IV), 1993.

[3] Arun Lakhotia. *A Workbench for Developing Logic Programs By Stepwise Enhancement.* PhD thesis, Case Western Reserver University, 1990.

[4] Leon Sterling and Marc Kirschenbaum. Applying techniques to skeletons. In J.M.J. Jacquet, editor, *Constructing Logic Programs*, chapter 6, pages 127–140. John Wiley, 1993.

# A Formal Model of Software Specification and its Automated Support

R. Mili
Kanata Software Engineering Services Inc.
75 Waterton Crescent
Kanata Ont. K2M 1Z2
email: rmili@csi.uottawa.ca
fax: (613) 599 8842

A. Mili
University of Ottawa
150 Louis Pasteur Private
Ottawa Ont. K1N 6N5
email: amili@csi.uottawa.ca
fax: (613) 564 5045

We use a mathematical relation-based model for the specification of software systems. A specification lifecycle is defined around this model, that allows for the systematic generation and validation of specifications. Automated tools for the generation and validation phases are discussed.

## 1 Specifications: The Product and The Process

### 1.1 The Product and The Process

As a *product*, a *specification* is a description of requirements that a user imposes on a software component he wishes to acquire. Such requirements may pertain to the functional properties of the component, or to such features as time and space performance, computing platform, software environment, etc...

As a *process*, a *specification* is best defined by its lifecycle. Like the overall software lifecycle, the specification lifecycle can be defined by means of its *phases* and its *activities* [2]. We identify two phases, namely the *generation phase* and the *validation phase*, and two activities, namely the *generation activity* and the *validation activity*. The two-dimensional structure that this defines for the specification lifecycle is given in figure 1.

### 1.2 Principles of Good Specification

In order to fulfill their function, specifications must satisfy a number of properties. We distinguish between properties of the product —which are intrinsic to the product and can be established by inspection of it, and properties of the process —which pertain to the relationship between the specification product and the user requirements.

We identify three properties of the product, namely: *simplicity*, i.e. being easy to understand by all parties concerned (the user, the programmer, the specifier, the domain expert, etc..); *formality*, i.e. being written in a formal notation whose interpretation is unambiguously defined; *abstraction*, i.e. being free of structural details and implementation biais. Further, we identity two properties of the process, namely: *completeness*, i.e. carrying all the user requirements; *minimality*, i.e. carrying nothing but the user requirements.

|  | Specification Generation Activity | Specification Validation Activity |
|---|---|---|
| Specification Generation Phase | Generating Specification | Generating Redundant Requirements Information |
| Specification Validation Phase | Updating the Specification Using V&V Feedback | Matching the Specification against Validation Information |

Figure 1: The Specification Process

## 2 Specifying with Relations

### 2.1 A Relational Model

We have found that the theory of relations, relying as it does on simple set theory (for the sake of simplicity and formality) and focusing on input output relations (for the sake of abstraction), is well adapted to the task of specifying software components. A specification is defined as the triplet $(X, Y, R)$, where $X$ (the input space) and $Y$ (the output space) are sets and $R$ is a relation from $X^*$ (the set of input sequences) to $Y$.

This model, which is akin to Mills' *black box specifications* [3] and to Parnas' *trace specifications* [1], encompasses the specification of simple input/output programs, abstract data type, and continuous processes. A specification $(X, Y, R)$ defines a simple input/output program whenever each output is defined solely on the basis of the current input (vs. the history of past inputs). Formally,

$$\forall x \in X, \forall Q', Q'' \in X^*, \forall y \in Y, (Q'.x, y) \in R \Leftrightarrow (Q''.x, y) \in R.$$

When the specification $(X,Y,R)$ defines a state-bearing software component, the internal space of the component (i.e. the set of internal states) can be defined as the quotient of $X^*$ by the equivalence relation $\xi$, where $\xi$ (which captures that two input sequences are interchangeable as far as the future is concerned) is defined as follows:

$$\xi = \{(Q, Q') | \forall q \in X^*, \forall y \in Y, (Q.q, y) \in R \Leftrightarrow (Q'.q, y) \in R\}.$$

### 2.2 A Lattice Structure

In our specification model, the refinement ordering is defined as follows: Relation $R$ is a refinement of relation $R'$ if and only if:

$$R'L \subseteq RL \wedge R'L \cap R \subseteq R',$$

where $L$ represents the universal relation on $Y$ and the concatenation represents the relational product. This ordering is interpreted as: any implementation that satisfies $R$ satisfies $R'$.

We have found this ordering to have lattice-like properties, whereby the *join* of relations $R$ and $R'$, when it exists, represents the specification that contains all the requirements information of $R$ and all the requirements information of $R'$; and the *meet* of relations $R$ and $R'$ represents the specification that contains all the information that is common to $R$ and $R'$.

## 3 Specification Generation and Validation

The lattice properties that we have uncovered above are the basis for formalizing the specification process, and automating it.

### 3.1 The Generation Activity

Given the very interpretation of the *join* operator of the lattice of specifications, it seems quite natural to envisage the following pattern of specification generation: the specifier identifies several distinct (not necessarily disjoint) aspects of the user requirements, focusses on each one of them in turn, and produces a relation for each. If the individual relations so obtained have a join (they dont always) then their join is taken as the overall specification; else the specifier goes back to the drawing board to double check his specification (or to check whether the user requirement is unsatisfiable).

To support this specification paradigm, we are designing a lattice based relational specification language where the specifier may write individual subspecifications in turn and have the language check that they have a join, and eventually compute it. This is done by compiling the specification language into a first order logic representation, which is in turn processed by a theorem prover [4].

### 3.2 The Validation Activity

As described in the specification lifecycle (figure 1), the validation activity spans two phases: the generation phase and the validation phase. In the generation phase, this activity consists of deriving redundant information about the user requirements, to be used subsequently to double check the generated specification. Two kinds of redundant information must be collected: *completeness properties*, which capture functional properties that the generated specification must have (but which the specifier may have overlooked); and *minimality properties*, which capture functional properties that the generated specification must *not* have (but which the specifier may have included inadvertently). The validation phase consists of proving that the generated specification is a refinement of all the completeness properties; and that it is not a refinement of any minimality property. Let $V_i$, $i = 1,2,3..$ be the relations that represent completeness properties, and let $W_j$, $j = 1,2,3..$ be the relations that represent minimality properties. In order to be complete (with respect to $V_i$) and minimal (with respect to $W_j$), relation $R$ must lie within the band shown in figure 2.

We have built a first prototype for computer assisted specification validation based on the proposed mathematical background and using Prolog. We are currently investigating a more powerful system, which relies on compiling specifications and properties into first order logic, which is in turn processed by a theorem prover [4].

**Acknowledgements**

The authors wish to acknowledge the contributions that Professor Roland Mittermeir, from the University of Klagenfurt, has made to the foundations of the proposed specifications model as well as to the progress of our work

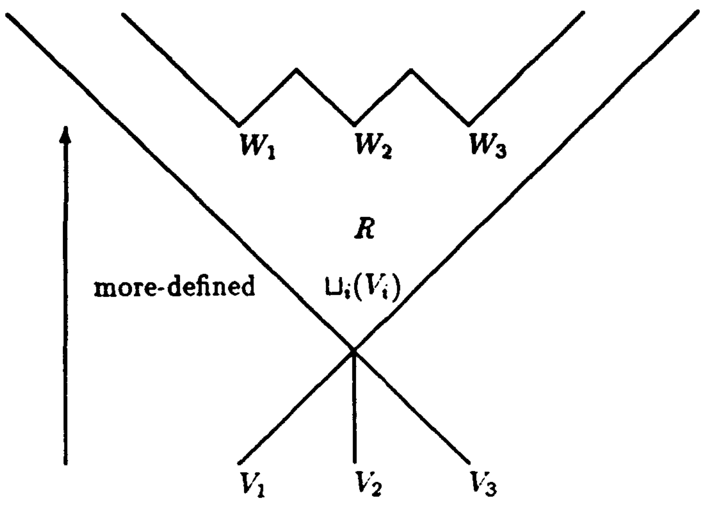


Figure 2: The Range of Complete and Minimal Specifications

## References

[1] Bartussek W. and D. L. Parnas. *Using Traces to Write Abstract Specifications For Software Modules.* UNC Report No. TR 77-012. December 1977.

[2] Boehm, B.W. *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice Hall, 1981.

[3] Mills, H.D., R.C. Linger and A.R. Hevner. *Principles of Information Systems Analysis and Design.* San Diego, CA: Academic Press, 1986.

[4] Wos, L., R. Overbeek, E. Lusk and J. Boyle. *Automated Reasoning: Introduction and Applications.* New York, NY: McGraw Hill, 1992.

# Evolution of Specifications

**Martin S. Feather** [*]
**USC / ISI, 4676 Admiralty Way, Marina del Rey, CA 90292, USA.** ***Email:*** **feather@isi.edu**

## 1 Introduction

The application of formal methods often makes the assumption that the object(s) being formalized and manipulated are likely to be behaviorally correct. For example, compilers increase efficiency while preserving the (supposed) correctness of their source code; the program transformation paradigm takes for granted the correctness of the specification, and works incrementally towards realizing effective and efficient implementations; stepwise refinement presupposes the starting point, and each intermediate point, to be a simple, correct, abstraction of the next step in the development; formal verification seeks to determine that a program denotes the same behaviors as a specification. The advantages of all these approaches stem from the relative easy by which a piece of software can be specified, as contrasted with the difficulty of realizing an efficient implementation of the same. However, attaining a correct specification may itself be a difficult task, perhaps more difficult than implementing it. Specifications are not simply the conjunction of requirements, but rather a compromise between conflicting requirements, a fleshing out of fragmentary requirements, a clear and organized description of jumbled, incoherent requirements. Formal methods can and should play a significant role in the derivation of specifications from requirements, with the understanding that this derivation process does not necessarily preserve functional behavior, but rather achieves compromises, generalizations, and reexpressions of behaviors.

The Software Sciences division at ISI, headed by Bob Balzer, has worked with program specifications for a long period [2]. A major thread of our research has been the study of an *incremental* approach to specification. In this position paper I summarize this thread, and mention some closely related efforts.

[*]Support for the author has been provided by Defense Advanced Research Projects Agency contract No. BAPT 63-91-K-0006; views and conclusions in this document are those of the author and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any other person or agency connected with them

## 2 Specification evolution at ISI

Our first foray into the incremental construction of specifications was Goldman's observation that natural language descriptions of complex tasks often incorporate an evolutionary vein - the final description can be viewed as an elaboration of some simpler description, itself the elaboration of a yet simpler description, etc., back to some description deemed sufficiently simple to be comprehended from a non-evolutionary description [8]. He identified three 'dimensions' of changes between successive descriptions: *structural* - concerning the amount of detail the specification reveals about each individual state of the process, *temporal* - concerning the amount of change between successive states revealed by the specification, and *coverage* - concerning the range of possible behaviors permitted by a specification.

Balzer went on to provide a complete characterization of generic changes to the structure of a domain model, and considered propagating the effects of those changes through the operations that use the model and through the already-established data base of information [3].

These papers established the key ideas of our work on the incremental construction / explanation / modification of specification, namely that:

- the increments of the process often *change* the specified behavior in a manner that is not necessarily a pure refinement
- there are different classes of such changes, and
- mechanical support is useful for conducting those changes over a large and detailed specification.

The main benefits provided by this incremental approach are those of:

*Specification Comprehension* — to gain an understanding of a large and complex specification, we may begin from a simple starting point, and thereafter incrementally introduce only a palatable amount of additional information at each step, leading ultimately towards the fully detailed specification. Furthermore, because we are not limited to pure refinements (as would be the case in the methodology of stepwise refinement), the earlier stages of the specification need not be restricted to pure abstractions of the later stages. This permits the use of 'white lies', as Balzer has termed them, namely simplifying assumptions that we know to be untrue, and that we will later retract, but which permit the expression of a readily understood idealized specification.

*Specification Construction* — analogously to comprehension, it is far easier to construct a specification incrementally than to compose the whole thing at once (the so called 'big-bang' fallacy of specification construction). We came to call our mechanization of these specification changes 'evolution transformations', to emphasize that theie very purpose is to evolve (change) the meaning of the specifications to which they are applied, in contrast to the more traditional meaning-preserving transformations that leave the meaning unchanged while changing some other aspect of the specification (typically efficiency or terminology).

*Specification Modification and Reuse* — one of the advantages of using specifications during software development is that they are more readily analyzable and maintainable (i.e., modifiable) than would be the corresponding efficient code. This permits the rapid exploration of alternatives during design. When analysis reveals that the specification needs to be modified, then evolution transformations ease this task. This is useful because, despite the advantages afforded by specification languages, specifications of complex systems can themselves be large and complex objects.

Our later work expanded upon these ideas, and is summarized next:

*Requirements and specifications* — We built a system, ARIES, for the support of acquisition of requirements and development of specifications to meet those requirements [10]. Central to this system was the notion that the early stages of software design are a very exploratory process, involving repeated cycles of examination and adjustment of the emerging specification. To make formal specifications palatable to the user, ARIES provided several 'presentations' (e.g., of data flow, type hierarchy) through which the user could view various aspects of specifications. These presentations were used both for display to the user, and as a medium through which the user could *modify* specifications — the user's direct manipulation of these presentations were used to retrieve from the library of evolution transformations those that would achieve the change the user had indicated [9].

*Parallel elaboration (specification merging)* — We extended the incremental development paradigm from a *linear* sequence of steps to a *tree* structure: starting from a single simple specification, different aspects of that specification were elaborated independently, leading to multiple specifications; these then were combined to achieve the all-inclusive specification [5]. This is very similar to the program merging of Reps et al and the software prototype merging work of Berzins et al [1]. Several key assumptions underly our particular style of merging:

- The parallel lines of incremental development are mostly independent. Thus incompatibilities arise relatively infrequently during combination; more commonly, combination of two lines of development gives rise to *additional* options from among which the user has to choose.
- Each of the specifications being merged has been derived from the single initial specification by a sequence of evolution transformations. As a result, some of the combination of the multiple specifications can be done by replaying the transformations in a linear order. Also, detection of clashes or further options that arise during combination can be determined by considering the transformation sequences that led to each of the specifications being combined [6].

## 3 Some related work

Fickas et al. have viewed specification development as a planning problem [1]; their *design operators* move within a search space of designs, transforming the specification as they do so. Interestingly, some of their work also uses a model of parallel elaboration (of requirements / specification) followed by combination, but in rather a different way: each elaboration is used to capture the ideal specification as seen by each different 'stakeholder' (e.g., intended user of the software system, administrator of the system, purchaser of the system). As a result, the ideal specifications that emerge from this process are very likely to be incompatible, and much of the combination process must deal with negotiation to reach an acceptable compromise among those divergent ideals [11].

The incremental approach to specification development is also being pursued by Souquières and Levy, who use their *development operators* (akin to our evolution transformations) to modify the emerging specification. Additionally, their development operators modify the *workplan*, an explicit record of the specification process, and modify the links between the emerging specification and that workplan [12]. This helps record the specification process itself.

[1]Hopefully reported on in detail at this workshop!

## 4 Further applications

To close, I mention a couple of areas where the incremental evolutionary approach may have a contribution to make, and that we are hoping to pursue further.

*Transformational development of distributed systems* — we would like to apply the program transformation paradigm to the development of distributed systems. To do this, we imagine commencing from an initial, idealized, *non-distributed* specification of the desired system, and proceed by the application of transformations that incrementally decompose the data and activities of the system across multiple components. However, one of the important ramifications of this application area is that we must abandon the premise that transformations are correctness preserving, in the face of the inevitable unreliability of the real distributed world. This suggests a blending of the more traditional kind of (correctness-preserving) transformational development with the issues of evolution transformations that we have been considering. We are pursuing a collaboration in this direction with Steve Fickas et al. at the University of Oregon.

*Expedient systems* — much of the complexity of real-world systems st^ms from the need to deal with issues such as misuse (unintentional or otherwise), component failure, and exhaustion of limited resources. Clearly, there is some relationship between an ideal system (one which has perfect users, unfailing components, unlimited resources, etc.) and an actual *expedient* system — one that embodies an appropriate compromise between ideals. Exploring the link between ideals and approximations of those ideals is an area that we think is worthy of further attention, and perhaps suitable for an evolutionary approach. Some limited experiments in this direction show promise — starting from global constraints (akin to integrity conditions), we can incrementally make them 'violatable', and introduce the code to detect and react to such violations: [4, 7].

## References

[1] J. Anderson and S. Fickas. A proposed perspective shift: viewing specification design as a planning problem. In *Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, USA*, pages 177–184. Computer Society Press of the IEEE, 1989.

[2] R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1267, November 1985.

[3] R. Balzer. Automated enhancement of knowledge representations. In A. Joshi, editor, *Proceedings, 9th International Joint Conference on Artificial Intelligence, Los Angeles*, pages 203–207, August 1985.

[4] R. Balzer. Tolerating inconsistency. In *Proceedings, 13th International Conference on Software Engineering, Austin, Texas, USA*, pages 158–165. IEEE Computer Society Press, August 1991.

[5] M.S. Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2):198–208, February 1989.

[6] M.S. Feather. Detecting interference when merging specification evolutions. In *Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, USA*, pages 169–176. Computer Society Press of the IEEE, 1989

[7] M.S. Feather. An implementation of bounded obligations. To appear in Proceedings, KBSE '93, The Eighth Knowledge-Based Software Engineering Conference, Chicago, Illinois, Sept., 1993.

[8] N. M. Goldman. Three dimensions of design development. In *Proceedings, 3rd National Conference on Artificial Intelligence, Washington D.C.*, pages 130–133, August 1983.

[9] W.L. Johnson and M.S. Feather. Building an evolution transformation library. In *Proceedings, 12th International Conference on Software Engineering, Nice, France*, pages 238–248. IEEE Computer Society Press, March 1990.

[10] W.L. Johnson, M.S. Feather, and D.R. Harris. Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering*, 18(10):853–869, October 1992.

[11] W.N. Robinson. Automated negotionated design integration: formal representations and algorithms for collaborative design. Technical Report CIS-TR-93-10, Department of Computer and Information Science, University of Oregon, April 1993.

[12] J. Souquières and N. Lévy. Description of specification developments. In *Proceedings of the IEEE International Symposium on Requirements Engineering, San Diego, CA, USA, January 1993*, pages 216–223. IEEE Computer Society Press, 1993.

# THE LIMITS OF FORMAL METHODS

(Extended abstract)

Alfs T. Berztiss

Department of Computer Science, University of Pittsburgh Pittsburgh, PA 15260, USA (alpha@cs.pitt.edu)

and

SYSLAB, University of Stockholm, Sweden

We introduce a classification of software into (1) basic data types, (2) data-carrying devices, (3) data transformers, (4) information systems, and (5) control systems. Software developers often take the view that the development of all software should follow the same pattern, and suggest their favorite specification methodology, be it VDM, or Z, or Larch, as the solution of all problems. On the other hand, special interest groups concerned with real-time systems, or information systems, or decision support systems, consider their application domain as all there is, and neglect to find out about approaches that have been found useful in other domains. Our purpose is to show that the level of formalism appropriate for different kinds of software differs, but that most application systems are made up of components belonging to all five kinds, and that there is no essential difference between programs and data bases. We further point out that in mathematics there are three levels of formalism: logic, "applied" mathematics, and statistics. Each has its own approach to validation: formal logical proof, informal mathematical reasoning, and experiment (what we call testing).

Software systems are being classified according to a variety of criteria. Our classification here regards all software as applications software and support software. We then classify components of application software according to purpose into information components, control components, and data transformers. An information component responds to queries, which are answered by examination of a persistent data base. The data base is subject to updates, and the maintenance of the integrity of the data base has to be addressed in system specification. Examples are a library catalog or an account management data base of a bank. Control components drive processes that are external to the controlling software. They are said to be embedded in the devices to be controlled. Here the specific concerns are temporal effects. Representative of control components are an elevator controller, an operating system, an air traffic control system, and a climate controller for a building. Examples of data transformers: a text formatter, a compiler, a spelling checker. Most application systems are hybrids containing components of all three classes. For example, the banking system maintains information about accounts (information activity), sends out overdraft notices (control activity), and prepares monthly statements for the customers (a rather trivial instance of data transformation).

In addition to the applications there are standard data types, such as integers, reals, and strings, and devices, such as stacks, arrays, and binary trees. We can think of all data transformers as operations belonging to standard data types or to devices. Then most application systems are essentially information-control systems that make use of standard data types and devices as the need arises. All such application systems can be regarded as embedded systems: systems with control as their primary function are embedded in machinery; systems that provide information are embedded in society - they receive inputs from society around them, and the information handed back by them is intended to affect the future of this society, i.e., the information systems exert some control on this society.

Let us look at data transformers. A data transformer can be considered in two ways. First, we take a type orientation. Under this orientation a particular set of objects and various functions associated with this set are regarded as a data type, and the concern is with individual elements of the set. For example, matrix multiplication is regarded as the generation of a new element of the set of matrices (the product matrix) from two existing elements of the set. The other is a stream orientation, and we speak then of a data stream transformer. This is a procedure that acts on a stream as a whole rather than on individual items. Data stream transformers accept data streams and generate outputs that are again data streams (e.g., a text formatter, a parser, a sorter, or a spelling checker) or that are much compressed (the count of zero elements of a matrix).

Advocates of software development based on data types generally think of the operations of a data type as functions, which leads to a classification of software according to the nature of the functions. The classes are defined by two orthogonal two-way partitions. The first relates to the determination of function values. The function telephone can be stored as a table of entries of the form ( subscriber, telephone number), for example (Berztiss, 624-8401), and one finds telephone(Berztiss) by looking up the table. On the other hand, the value of cosine for a given angle is found by application of a computational rule to the angle. The other partition separates functions into static or immutable and dynamic or mutable. An immutable function does not change - for example, the cosine function, or a finite function that supplies the times of sunrise at Pittsburgh for the 365 days of 1997. A mutable function changes. Thus, before I got my present office, telephone(Berztiss) had the value 624-6458. Now the value is 624-8401. However, the sunrise function can be implemented in two ways: as a piece of code that computes the time of sunrise for a given date, or as a a table with 365 entries. Hence, at a sufficiently high level of abstraction there is no difference between a program and a section of a data base.

Now, the support software, i.e., standard data types and devices, can be very well specified in a formal way, and logical proofs are appropriate for this type of software. In this context the proofs are unlikely to become excessively long. There is another group of software for which logical proofs can be constructed. This is software that is specified by the software developers themselves, because here validation is no more than showing that the software satisfies its specification. But this holds only as long as the software specifiers in fact understand their specifications, and unreadable formal specifications arise more often than one may think.

The real problems arise with software that contains conceptual models of the domains in which it operates. Here a specification is the point of contact between developers and domain experts, and the domain experts must be able to read the specification to determine that it has captured their expectations. The form of the specification is not required to support logical proofs, but it should be sufficiently rigorous to prevent any ambiguity. Moreover, reasoning of the type found in mathematical proofs should be possible, e.g., it should be possible to prove that the specification has captured the informal requirement that the doors of an idle elevator that has been moved to a holding floor are to be closed. Numerous examples of mathematical proofs are given in [1].

## References

[1] Berztiss, A., Programming with Generators: An Introduction. Ellis Horwood, Chichester, England, 1990.

# FORMAL SPECIFICATION OF THE SOFTWARE PROCESS

(Extended abstract)

Alfs T. Berztiss

Department of Computer Science, University of Pittsburgh Pittsburgh, PA 15260, USA (alpha@cs.pitt.edu)

and

SYSLAB. University of Stockholm, Sweden

Reengineering of organizations is a term that is being heard more and more often, with three books on business reengineering having appeared this year alone [1, 2, 3]. Its purpose is to improve the performance of an organization by at least a factor of two, in terms of the time to perform a task, in terms of costs, or in terms of the quality of a product. This is achieved by considering the mode of operation of the organization as a set of processes, and by improving each process. In our context an organization can be a business corporation, a government office, or a military command.

Although there is no recipe for reengineering, there is a preferred approach. This is to entrust the reengineering of an organization to software engineers. They should be experienced in analysis of processes, and they should be familiar with notations and tools that allow process descriptions to be transformed automatically into software systems to support the operation of the reengineered organization. However, the software engineers will have to implement changes much more radical than the changes they have had to deal with in the past. In fact, in order to deal with these changes, they will have to reengineer their own software development processes first of all.

Unfortunately few of the people who regard themselves software engineers are capable of carrying out any complex reengineering task, for which various reasons have been given [4]. My contention is that the primary reason is an unwillingness or inability to take a formal approach. Another major reason is the attitude that there is a single software development process that is to serve every software development effort. Indeed, there is no fundamental difference between a business process, which is essentially a service activity, an industrial manufacturing process, and a process embedded in a manufactured product to control its operation. They can all be defined in terms of formally defined operations associated with data types. But decisions have to be made as to which operation is to be applied when. Hence a process is made up of decision centers, and each decision can be made by an unassisted human, by a human assisted by a decision support system, or by the system itself. The exact way the decisions are made defines a variety of process structures. Still, taking a formal approach allows a process to be designed and validated without regard to its ultimate implementation. This means that once the essential nature of a process has been established, the process can be implemented under a traditional organizational structure or a reengineered structure. In other words, the essence of a process is independent of the structure of the organization that supports the process. Summarizing: a process has the same basic structure independent of the domain of application; the process is a sequence of decision steps; the process can be defined independently of any organizational change introduced to improve the execution of the process.

We have used the specification language SF (Sets-Functions) [5, 6] to express formal definitions of processes, and, in particular, the software process itself. An SF specification consists of one or more segments. Each segment has three components: a schema definition, specifications of events, and a control component called the responder. A segment corresponds to a data type. The schema definition identifies a set of interest for the segment, e.g., a set of bank accounts or a set of persons. The schema definition also introduces a set of functions (finite maps). Examples of functions for bank accounts: balance in an account, transactions for the current month (a set-valued function). Some functions have a null domain - these constant function represent properties that pertain to the entire segment, e.g., interest rate. The schemas of all the segments of a system may be regarded as defining a data base.

The SF sets and functions are mutable, and events are operations that change them. An event may be initiated by a user, e.g., by depositing money into an account, or by the system itself, e.g., the calculation and crediting of interest at set time intervals. Events may be provided with preconditions, which ensure that an event will take place only if all its preconditions are true. Events may raise signals.

The responder consists of transactions, and each transaction is associated with a time indicator. A transaction is activated by a clock, one or more signals, or a combination of clock and signals. If a signal alone is to activate a transaction, the time indicator is set to the value of a function that always returns the actual time - then, as soon as the signal that can activate the transaction is raised, the transaction is activated. An example of a transaction activated by the clock alone causes all doors of a building to be unlocked at 7:00 each morning. In a different version of this transaction the unlocking would take place only if a signal had been set at some earlier time.

A transaction can act in three different ways. First, it can initiate an event. For example, in a library, after the library closes for the night, a transaction initiates an event for every outstanding book that should have been returned on that day, and the event marks this book overdue. Second, in a situation where the system knows that an event is to be initiated, but cannot supply the inputs to this event, a transaction issues a prompt. This we discuss further down. Third, the transaction may merely remind users of something or other, e.g., to send out an acknowledgement that the paper discussed above has been received.

What we have here is that an event raises a signal, and the signal is picked up in this or some other segment. This becomes interesting when the transaction initiates another event, this event raises another signal, the signal is picked up, and so forth. We have then a facility for defining processes - SF processes correspond to Petri nets with events corresponding to places, transactions to transitions, and signaling to the movement of tokens.

Let us now look at transactions that issue prompts. Consider the arrival of a paper at the editorial office of a journal. Three referees are to be selected, and there are three options: (i) an editor selects the referees without any support by the system; (ii) the system helps the editor select the referees, by supplying the editor with information the system deems relevant; (iii) an expert system is developed that takes over the referee selection. Every prompting transaction is a decision center, and the prompting transactions define all the opportunities for process automation that there exist. The automation can be carried out incrementally, based on a priority scheme determined by cost-benefit analysis carried out for each prompting transaction.

## References

[1] Davenport, T.H., Process Innovation: Reengineering Work through Information Technology. Harvard Business School Press, Boston, MA. 1993.

[2] Johansson, H.J., McHugh, P., Pendlebury, A.J., and Wheeler, W.A., Business Process Reengineering: Breakpoint Strategies for Reengineering. Wiley, Chichester, England, 1993.

[3] Hammer, M., and Champy, J., Reengineering the Corporation: A Manifesto for Business Revolution. Harper Business, New York, 1993.

[4] Yourdon, E., Decline and Fall of the American Programmer. Yourdon Press, Englewood Cliffs, New York, 1992.

[5] Berztiss, A.T., Data abstraction in the specification of information systems. Proc. IFIP World Congress 86, pp. 83-90.

[6] Berztiss, A., Formal specification methods and visualization. In Principles of Visual Programming Systems. S.-K. Chang (Ed.), Prentice-Hall, Englewood Cliffs, NJ, 1990, pp. 231-290.

# State-Based Specifications In-The-Large*

Alan C. Shaw
Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195
*shaw@cs.washington.edu*

## 1. Introduction

Our goal is to provide mechanisms for the specification of software requirements and designs for large real-time systems. Among other features, these mechanisms should be executable, universal, formal, and scalable. The basis for our work is the communicating real-time state machine (CRSM) notation [5, 6, 7, 8]. CRSMs are universal state machines with guarded commands as transitions, synchronous IO communications over unidirectional channels, and facilities for describing the execution times of transitions and for accessing real-time. CRSMs are distinguished from other state machine models mainly by their explicit timing features.

In this position paper, we propose both an architecture and a particular set of in-the-large paradigms for specifying real-time software. Related work includes statecharts with their superstates, series/parallel compositions, shared storage model, and broadcast communications [3]; IO automata with its broadcast communications and nice formal composition [4]; and programming notations such as timed CSP [1] and CSR [2] that use CSP-like synchronous communications, deal with time explicitly, and provide some abstractions for larger objects.

## 2. Behaviors, Components, and Interfaces

A real time system is treated as a closed world consisting of an external environment and a controlling and monitoring computer system; the environment and computer system communicate through IO signals (events, messages, commands,...). The behavior of a real-time system is characterized by the set $T$ of *traces*, where a trace is a (possibly infinite) sequence of timed IO events. A timed IO event is a triple, (event_name, event_value, time). For correct specifications, we assume that $T_R \supseteq T_D \supseteq T_I \supseteq T_E$, where the $T_i$ are the traces for the requirements, design, implementation, and execution, respectively.

The environment and the computer system are each described by a set of CRSMs and their communicating IO channels. A connected channel has a name, a message type, a sender CRSM, and a receiver CRSM.

IO channels are the interfaces among CRSMs and among subsystems of CRSMs. Unconnected channels are those with either an undefined sender or an undefined receiver, and represent possible interfaces to other components (yet to be connected). System behaviors are defined by the traces over their connected channels. Our in-the-large component is either a single CRSM or a

* This research was supported in part by the National Science Foundation under grant number *CCR-9200858*.

set of CRSMs. For most applications, we expect the set to be structured as follows.

## 3. Controller-Client (C-C) Architecture

First, we assume a standard interface for a basic reusable CRSM, consisting of a *start* channel for initiating execution, a *stop* channel for signaling termination, and an arbitrary number of other IO channels. Both *start* and *stop* may have message parameters.

A higher-level component has the same standard interface, and contains a *controller* machine and any number of *client* subsystems (sets of machines). The clock machine* of the controller is used as the clock machine for the higher-level component; similarly, the *start* and *stop* channels of the controller are the *start* and *stop* channels for the component. The tasks of the controller are to initiate clients, control their execution, and synchronize their termination.

System composition and refinement are defined in terms of this C-C architecture. A set of subsystems are composed into a larger component by providing an appropriate controller. The refinement of a C-C subsystem is its set of clients.

## 4. Standard Compositions and Date Encapsulations

Conventional control schemes for composing objects are defined by C-C structures. In each case, the implementing controller can be given by a CRSM.

*Sequential* and *parallel* combinations are both straightforward. The parallel case corresponds to a fork and join of the controlled client subsystems. *Functional* composition, where the clients specify mathematical functions, is a special case of the sequential connection.

Slightly more complex is *guarded selection*, a control structure that selects one of a given set of clients for execution based on the values of Boolean guards. Repeated execution of a client while some guard remains true - an in-the-large *while* loop - is yet another useful conventional control form.

Shared data in our distributed model can be specified with *abstract data type* (ADT) components. At the lowest level, a shared data server with read and write channels can be described easily by a single CRSM. For higher levels, a server for an ADT implements general operations (procedures, methods, ...). A send/receive protocol over the IO channels defines the user interface of the ADT for each operation; the controller interprets the protocol to invoke the operation (client) that is requested.

## 5. Real-Time Paradigms

For specifications involving interrupts, faults, and time constraints, we propose several higher level utilities and control structures.

* Every CRSM has an associated real-time clock machine and connected channel, which provide global real-time and an interval time-out on request.

Two types of time-constrained utilities appear in many of our real-time specifications. The first is *alarm clocks*, which work at a higher level than our real-time clock machines. Alarm clock CRSMs are defined for both relative and absolute time - the former generates a *wakeup* message at a given interval and the latter generates a wakeup at some future absolute time. Parameters to these machines include a *reset* and a *ring time* that determines how long the alarm will be enabled after the wakeup is first triggered.

The second utility is a *broadcast* or *multicast* CRSM family, . that broadcasts messages to some specified subset of system components. The facility allows a message or signal to be broadcast for a given time interval; during the interval, any of a set of designated receivers can elect to receive the message.

We borrow an idea used in statecharts in order to handle software and hardware *interrupts*. A transition that exits and reenters an entire CRSM or a component is semantically equivalent to that transition connecting every state in each CRSM of the component to its start state. The multicast facility described above is used to construct the higher-level version that ensures that all machines in a subsystem are (gracefully) interrupted.

From requirements through implementations, and from theory through practice, the principal ways to organize real-time functions and tasks are as *periodic* and *sporadic* (event-driven) activities. For most real-time tasks, we propose a C-C structure. A periodic controller activates its client task periodically and generates a timing fault interrupt if a deadline is exceeded; the period and deadline are parameters of the controller. Similarly, a sporadic task controller has event and deadline inputs; when the triggering event is received on its IO channel, the controller initiates its client sporadic task and generates a timing fault interrupt if the deadline is exceeded.

Finally, we note that time-constrained sharing of resources that are required by components can be included in the specifications. Higher-level processor sharing among a set of independent components can be handled through an "executive" controller that schedules clients, say in round robin fashion. Sharing of communications lines, input-output devices, and channels, can be accomplished by defining a CRSM and shared channels to simulate the actual sharing.

## 6. Discussion

There are natural textual and graphical representations for (almost) all of the structures proposed here. The textual forms are fairly obvious. Graphically, we represent each CRSM and higher-level component by a labeled rectangle with rounded corners; each channel is drawn as a labeled wavy arrow (denoting communications) from its sender to its receiver. Facilities defined by C-C architectures can also be represented by a appropriate icon for the controller with connecting arrows to clients. We need to build some software tools that permit the construction, editing, and testing of these representations.

Several naming issues also arise. It should be possible to connect type-compatible channels with systematic renaming of sender and receiver IO calls. Arrays of channels are clearly necessary for "server" subsystems, e.g., broadcasting, ADTs, n-way fork/join controllers. Class-instance declarations also need to be provided. All of this seems no different than analogous facilities in modern programming languages and can be directly borrowed.

Finally, beyond specifying in-the-large, we would like to monitor and verify behaviors in-the-large. This requires some computer tools and thinking beyond our present work.

We have presented some ongoing research that attempts two things - first, to define the types of components needed for real-time specification in-the-large and second, to show who these components may be constructed using communicating real-time state machines.

References:

[1] J. Davies and S. Schneider, "An introduction to timed CSP," *Tech. Monograph PRG-75*, Oxford University Computing Laboratory, Aug. 1989.

[2] R. Gerber and I. Lee, "A layered approach to automating the verification of real-time systems," *IEEE Trans. on Software Eng.*, Vol. 18, No. 9, Sept. 1992, pp. 768-784.

[3] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming 8*, 1987, pp. 231-274.

[4] N. Lynch and M. Tuttle, "An introduction to input/output automata," *CWI - Quarterly 2*, 1989; also, *Tech. Memo. MIT/LCS/TM-373*, Laboratory for Computer Science, MIT, Nov. 1988

[5] S. Raju, "An automatic verification technique for communicating real-time state machines," *TR #93-04-08*, Dept. of Computer Science & Engineering, University of Washington, Seattle, April 1993.

[6] S. Raju and A. Shaw, "A prototyping environment for specifying, executing and checking communicating real-time state machines," *TR #92-10-03*, Dept. of Computer Science & Engineering, University of Washington, Seattle, October 1992. A revised version is in publication in the journal *Software-Practice & Experience*.

[7] A. Shaw, "Communicating real-time system machines," *IEEE Trans. on Software Eng.*, Vol. 18, No. 9, Sept. 1992, pp. 805-816.

[8] 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.

# Designing and Specifying Flexible Concurrency Control: Position Paper

David Stemple

Computer Science
University of Massachusetts
Amherst, MA 01003

Effective specification technology must facilitate the writing, reading and understanding of specifications by humans as well as enable mechanical reasoning to prove desirable properties of specifications. Encapsulation and abstraction mechanisms, similar to those of programming languages, can help meet both of these goals. Types can be used to encapsulate algebraic specification modules in ways that are easily understood by human designers, and higher order functions can be used to produce abstractions that organize designers' thinking and provide powerful hooks for mechanical reasoning.

In addition to using encapsulation and abstraction, it may be best to start the specification task by designing an abstract operational model having formal underpinnings that facilitate the verification of important properties. These properties then become the specification of the system at hand. The major benefits that can accrue to this approach are easier verification of specification properties, assurance that a specification has a model, quicker prototyping, and better human engineering of the specification process. The approach contrasts with techniques where a predicate based specification of system invariants is produced before operational models or designs.

We have used this approach for the specification of database systems where our main goal was the design of transactions that provably obeyed all integrity constraints, thus obviating the need for any run-time check beyond that contained in transaction code and type checking of input values. We have now turned to the task of specifying flexible concurrency control systems.

Coordinating a set of computations that share data is a complex undertaking. Mechanisms for such coordination have been designed for operating systems, programming languages and database systems. These include semaphores, monitors, mutual exclusion, path expressions, locks, and optimistic concurrency control systems. It has proven difficult to characterize the power and behavior of these mechanisms and to compare them with each other. This difficulty stems from both the low level nature of the mechanisms and the inherent complexity of the problem. We are concerned with simplifying and clarifying the task of building systems for controlling the coordination of computations on shared data in a persistent programming environment.

The goal of this approach is to control the coherence of sequences of operations on shared data, called actions. An extreme example of the coherence of a sequence of operations is the atomic transaction in which the operations are isolated from the effects of any other concurrent transaction and only allowed to have an effect on the database as a single atomic unit. The actions are programmed by specifying algorithms for manipulating the data and by annotating the algorithms with markers that specify how the data sharing is to be controlled. In systems supporting atomic transactions the strong coherence can be programmed simply by inserting begin and end transaction markers around a set of database operations to delineate an atomic transaction. The effect of begin and end has to be specified separately at another level in the system. Examples of more complex coherence occur in design systems where parts of a design are updated locally by multiple designers and the different changes are reconciled before their joint effect is made to the global design. Many ways of achieving the isolation of atomic transactions have been devised and implemented and many mechanisms for coordinating cooperative computations have been proposed. Our main purpose is to present an approach to designing concurrency control in which multiple schemes beyond serializable models can be specified in a manner that is understandable and supportive of implementation efforts. We call the model embodying our approach the Communicating Actions Control System or CACS.

The CACS language is a *formal design language supporting mechanical theorem proving*. The small set of structures of the CACS abstract machine are themselves algebraic abstract data types. The manipulation of these structures in the language is controlled in a way that is designed to ease the problem of verifying design properties. The main technique for this is to design control structures and complex instructions around a few higher order functions. Thus the CACS abstract machine is an example of using a small set of abstractions and encapsulations - the few types, control structures and complex instructions - to facilitate the production of understandable, tractable specifications.

The style of specification in CACS is to first develop an abstract model design and then to verify its properties, the system invariants. This twofold activity produces a set of invariants like those that constitute the first phase of many specification approaches, but also a formal, abstract, behavioral design that provides a model of the invariants. This not only provides a double view of a concurrency control scheme, but shows productively that the invariants are satisfiable. Using feedback during attempts to prove invariants of a CACS design increases the probability that the design has its intended semantics. Assuring that a design has its intended semantics is obviously a formally unattainable goal but one that should nonetheless be strived for by a specification technique. A hypothesis of the CACS experiment is that a combination of an abstract model approach, verified invariants and the feedback provided during the phases of specification can lead to a usable technology for the meaningful and useful specification of concurrency control schemes.

In CACS the initiation of control behavior is specified using event patterns in rules. The controlling behavior itself is specified in rule bodies using a version of the abstract model technique: behavior is specified by its effect in terms of an abstract model, a mathematically understood model such as that defined by set theory. The CACS abstract model comprises structures (tuples), lists, finite sets and enumerated, mutable functions (data functions). The core of CACS behavior is specified in terms of changes to states constructed using these four types. The semantics of the types are themselves based on four sets of algebraic specifications. Sets of theorems proven from these algebraic specifications can be used to support mechanical reasoning about CACS design properties. In this way CACS layers an abstract model approach on top of an algebraic specification level and delivers the power of reasoning about the predefined algebraic specifications to the problem of verifying properties of individual CACS designs. A secondary benefit of the approach is the elimination of the need for writing algebraic specifications and its replacement by the opportunity to specify abstractly and formally using manipulations of an abstract model.

A CACS design defines four major elements of a control system: the kinds and sequence of events that can occur in the event stream, the patterns of events that will initiate controlling behavior, the structure of the state used in controlling the system, and the way in which the state of the system changes in response to the events. The CACS specifications of these elements are formal and together support *mechanical proofs of properties of* individual designs. The interesting properties of designs include invariants on the state of the controlling system, temporal relations between the state and events in the event stream, and invariants on the event stream. Two styles of reasoning are to be used in verifying CACS design properties and it is an experimental hypothesis of the CACS experiment that using the two styles rather than one will lead to tractable reasoning. The two styles derive from the fact that the rule bodies that specify state changes have one formal capture and the rule patterns, rule order and event stream that together determine the sequence of rule firings have a different formal capture. Thus proving properties of CACS specifications will involve multiple stages of the different kinds of reasoning. State invariants of rule bodies will be verified by heuristically rewriting functional expressions and temporal relationships will be shown by constructing chains of causality across rule patterns.

# Testing by Narrowing

Extended Abstract

Sergio Antoy and Dick Hamlet

Portland State University
Department of Computer Science
Portland, OR 97207
{antoy,hamlet}@cs.pdx.edu

## 1 Introduction

Testing and debugging a program $P$ may require computing an input $I$ such that the execution of $P$ on input $I$ goes through some given path $T$ of $P$. We describe how to compute such an input for programs coded in a simple imperative language with generic expressions including user-defined abstract data types.

For example, consider the following program which computes iterative a preorder traversal of a tree. *Stack* and *tree* are user-defined types.

```
declare s : stack; t : tree;
begin
  if not(is_null(t)) then
    s := empty;
    push(t, s);
    while not(is_empty(s)) loop
      declare x, y : tree;
      begin
        y := top(s);
        pop(s);
        visit(y);
        x := left(y);
        if not(is_null(x))
          then push(x, s);
        end if;
        x := right(y);
        if not(is_null(x))
          then push(x, s);
        end if;
      end;
    end loop;
  end if;
end;
```

We may wish to compute an input that leads to the execution of two iterations through the body of the *while* statement such that during the first iteration the guard of the first *if* statement fails and the guard of the second *if* statement succeeds, whereas during the second iteration these conditions are reversed.

Problems of this kind are unsolvable in general. However, the technique that we describe is capable of finding any existing solution to the problem.

Our technique is a two-step procedure. First, given a program $P$, we compute the weakest precondition [4], say $W$, that guarantees the execution of a given path $T$ of $P$. This computation is straightforward. Second, we attempt to solve the equation $W = true$ with respect to the variables of $P$. Operationally we use narrowing [5], a sound and complete procedure for solving equations involving symbols defined by a term rewriting system [6]. For our specific problem, if the narrowing procedure finds a solution $I$, then $I$ is an input to $P$ that executes $T$, and conversely, if there exists an input $I$ to $P$ that executes $T$, then the narrowing procedure finds $I$ as a solution.

Next we describe in some detail the two steps of our technique, we outline a prototypical implementation, we discuss how more difficult problems can be solved, and finally we briefly relate our approach to similar ones previously proposed.

## 2 Weakest Precondition

Given a program $P$, a condition $C$, and a path $T$, we compute $W = wp(P, C, T)$, the weakest precondition such that the execution of $P$ begun in any state satisfying $W$ goes exactly through the statements of $T$ and leaves the program in a state satisfying $C$. $wp$ is a standard predicate transformer for a deterministic language [8], except for the presence of a third argument, the path $T$.

Such a path must be statically plausible, i.e., it could be executed if we were allowed to arbitrarily change both the state and the constants in the program before the execution of each statement. This condition is easy to verify. The third argument of *wp* controls the choice of the next statement of a branching statement. A weakest precondition with respect to a fixed path is computed more easily than the general weakest precondition of a program.

Weakest preconditions may be huge expressions even for simple problems. The weakest precondition of the tree traversal problem discussed earlier is *and(not(is_null(t)),...)*, where there are a total of 75 occurrences of the program's operations and the variable $t$. In the following, we will denote this expression with $\mathcal{W}$.

## 3 Narrowing

The meanings of both the predefined operations of a language and the user-defined operations of a program are given by an equational specification. If $l = r$ is an equation, we stipulate that in an expression we can replace an instance of $l$ with the corresponding instance of $r$, but not vice versa. For this reason we rather write our equation $l \rightarrow r$ and call it a *rewrite rule* [6].

For example, all the operation symbols occuring in $\mathcal{W}$, whose computation was discussed earlier, are specified as follows. The signature is obvious from the context. Capital letters stand for variables. The symbol '_' represents an anonymous variable.

$$\begin{aligned}
and(true, X) &\rightarrow X \\
and(false, \_) &\rightarrow false \\
not(true) &\rightarrow false \\
not(false) &\rightarrow true \\
is\_empty(empty) &\rightarrow true \\
is\_empty(push(\_, \_)) &\rightarrow false \\
pop(push(\_, S)) &\rightarrow S \\
top(push(E, \_)) &\rightarrow E \\
is\_null(null) &\rightarrow true \\
is\_null(tree(\_, \_, \_)) &\rightarrow false \\
left(tree(\_, L, \_)) &\rightarrow L \\
right(tree(\_, \_, R)) &\rightarrow R
\end{aligned}$$

A narrowing step of an expression such as $\mathcal{W}$ consists in computing a reduct of $\sigma(\mathcal{W})$, where $\sigma$ is a substitution for the variables of $\mathcal{W}$ such that $\sigma(\mathcal{W})$ is reducible. For example, we may instantiate $t$ to $tree(x, y, z)$, where $x$, $y$, and $z$ are arbitrary values and reduce $is\_null(tree(x, y, z))$ to *false*. Thus, we solve an equation such as $\mathcal{W} = true$ by narrowing $\mathcal{W}$ all the way to *true*.

A solution for $\mathcal{W}$, computed by narrowing, is $t = tree(u, null, tree(v, tree(w, x, y), null))$, where $u$, $v$, $w$, $x$, and $y$ are variables.

A brute-force implementation of narrowing is generally very inefficient. Relatively efficient implementations are based on strategies that limit the number of substitutions and positions that must be considered in a narrowing step. These strategies preserve the completeness of narrowing for the rewrite systems that we generally obtain in specifying the data types used in programming.

## 4 Implementation

Our prototypical implementation of our technique is coded in Prolog. The implementation comprises two major modules. One computes weakest preconditions and the other attempts to narrow them to *true*. The first module is small and conceptually simple, since the formal definition of our predicate transformer is easily mapped to Horn clauses. The translation of a program from its usual form to the form expected by the implementation of the predicate transformer is a non-trivial, language-dependent problem that we have not undertaken yet.

The second module is small too, but conceptually more complex. Several interesting implementation issues arise, in particular, the completeness of the narrowing procedure and the efficiency of the computation. We implement a lazy strategy that takes advantage of a particular representation of the rewrite rules and we couple our strategy with a breadth-first control regime.

Narrowing steps are "don't know" non-deterministic, whereas rewrite steps are, to a large extent, "don't care" non-deterministic. Thus, we gain efficiency by repeatedly reducing the needed redexes of an expression that is to be narrowed. Since some of these expressions may not have a normal form, some care must be taken to preserve the completeness of our implementation.

## 5 Advanced features

Statically plausible paths may be semantically impossible. For example, this is easy to see for the tree traversal program. Every time that *push* is invoked one iteration through the loop body must also be executed before the loop terminates. Thus, we may specify a path $T$ that is not executed for any input. In this case, the resulting weakest precondition $W$ has no solutions. If we try to solve $W = true$ by narrowing, the computation may or may not terminate. In practice, we fail to find inputs that reach the statements following a *while* statement.

To overcome this problem we weaken the constraints imposed on the path through a *while* statement. We specify how many iterations are executed through the body, but do not specify the path of each iteration. A weakest precondition for such a less-constrained path is easily computed using power functions [1]. If $f : S \rightarrow S$ is a function on the state $S$ of a program, the power function of $f$, $f^* : \mathbb{N} \times S \rightarrow S$, is defined by

$$f^*(k,s) = \begin{cases} s, & \text{if } k = 0; \\ f^*(k-1, f(s)), & \text{if } k > 0. \end{cases}$$

This approach allows us to find inputs to a program for reaching the statements that follow a loop.

For example, suppose that the problem is to find an input to the *while* statement of our tree traversal program that leads to the termination of the loop after exactly three iterations through the body. First, we compute the following weakest precondition, where *power* denotes the power function of the functional abstraction of the loop body.

$$\begin{array}{l} \mathit{and}(\mathit{not}(\mathit{is\_empty}(\mathit{power}(0,s))) \\ \quad \mathit{and}(\mathit{not}(\mathit{is\_empty}(\mathit{power}(1,s))), \\ \quad\quad \mathit{and}(\mathit{not}(\mathit{is\_empty}(\mathit{power}(2,s))), \\ \quad\quad\quad \mathit{is\_empty}(\mathit{power}(3,s)) \end{array}$$

Second, we solve this condition by narrowing, and we discover that there are only 10 distinct solutions: 5 in which $s$ initially contains 1 tree only, 4 with 2 trees, and 1 with 3 trees.

For each solution we can find the path executed within the loop body by symbolic execution or by profiling an actual execution.

## 6 Related work

There are two research areas related to our work: program analysis and narrowing. The problem of finding inputs that will force the execution of a path in a program has been attacked in a number of ways. For example, [2] used symbolic execution and a linear-programming equation solver. Our approach differs from previous work in several significant ways:

1. Whereas it is usual to convert programs into control-flow graphs and compute paths in these graphs, we instead use a linear notation based on the program syntax itself.
2. Whereas it is usual to implement symbolic execution by following a flowgraph path from top to bottom, we instead use the weakest-precondition formalism that proceeds from bottom to top.
3. Whereas the usual method of solving equations is to employ some form of linear programming or matrix manipulation, we use narrowing.

The first two distinctions are matters of style only; nothing can be done using our approach that cannot be done in the previous way. Nevertheless, we believe that our approach is simpler. The final distinction is one of substance; narrowing can not only handle the numerical data types that previous solvers could, but can in principle deal with equations using arbitrary abstract types, and equations that arise from additional constraints. Thus we expect that our approach applies to a far wider class of programs, and to wider and more difficult problems.

Narrowing is the operational principle of languages that integrate the functional and logic paradigms [3]. Our application is unorthodox and the closest related work concerns the implementation of narrowing in Prolog, e.g., [7]. For our application the completeness of narrowing is crucial, thus, we have extended previous approaches by replacing the default depth-first search strategy with a breadth-first one and by interleaving narrowing with extensive rewriting.

## References

[1] S. Antoy. Automatically provable specifications. Technical Report 1876, Dept. of Computer Science, University of Maryland, 1987.

[2] L. Clark. A system to generate test data and symbolically execute test programs. *Trans. on Soft. Eng.*, SE-2:215–222, 1976.

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

[4] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. of the ACM*, 18:453–457, 1975.

[5] M. J. Fay. First-order unification in an equational theory. In *Proc. 4th Workshop on Automated Deduction*, pages 161–167, Austin, TX, 1979. Academic Press.

[6] J. W. Klop. Term Rewriting Systems. In S. Abramsky et al., editor, *Handbook of Logic in Computer Science, Vol. II*, pages 1–112. Oxford University Press, 1992.

[7] R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. PLILP'93*. Springer LNCS, 1993. (To appear).

[8] R. T. Yeh. Verification of programs by predicate transformation. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, volume 1, pages 228–247. Prentice-Hall, 1978.

# Finite-State Verification and Software Design

**Rance Cleaveland**
**Department of Computer Science**
**North Carolina State University**
**Raleigh, NC 27695-8206**
**USA**
**tel: (919) 515 7862**
**fax: (919) 515 7896**
**e-mail: rance@csc.ncsu.edu**

September 14, 1993

**Abstract**

This note argues for the utility of finite-state specification verification methodologies in the analysis of software design specifications.

## 1 Introduction

The past ten years have witnessed a surge in research on automatic techniques for establishing that finite-state systems satisfy their specifications[6, 8, 12, 15]. This area of research has strong practical motivations; systems whose correctness has been established formally are in some sense free from error, and the purely automatic nature of the methods under investigation removes the burdens that traditional non-automatic formal methods place on system builders. The domains to which these methods have traditionally been applied include hardware design, communications protocols, and process control systems. However, recent work indicates that software requirements specifications and designs [1, 2] can also be formulated in ways that would benefit from such analyses. This note suggests how finite-state specification and verification techniques may be brought to bear on the problem of software design and advocates further collaboration between researchers in software engineering and automatic verification.

## 2 Event-Based Specifications

Parnas and others [11] have suggested the use of an "event-based" style of software requirements specification. In this style, one formulates the requirements of a system in terms of a finite-state machine that describes the changes that a system undergoes in response to different inputs given it by its environment. Several different nontrivial systems have been specified in this style; noteworthy examples include work related to A-7 fighter aircraft project [2] and Leveson et al.'s landmark work [14] on the specification of air-traffic control software.

A chief virtue of such specifications is that, because they are formulated in precise mathematical terms as finite-state machines. techniques for analyzing analyzing finite-state systems may be brought to bear on them. Thus, design properties may be formally established without the expense of implementing prototypes. At one level, for instance, the specifications may be rigorously exercise using finite-state system simulators (such as those provided by the StateMate tool [9]); at another, specific properties such as deadlock-freedom may be formulated as temporal logic formulas and tested against specifications using model checkers [1, 2]. Specifications may even be compared to determine if they conform to one another.

Another virtue of event-based specifications is the existence of languages for the definition of finite-state event-based systems and associated analysis tools. Languages such as LOTOS [4] and ESTELLE [5] have been developed to facilitate the description of communications protocols, while ESTEREL [3] and STATECHARTS [10] support the structured development of synchronous systems used in process control. Typically, these languages include constructs for the modular design of systems; these structuring facilities would also be useful in the development of high-level software designs, since they can provide a formal basis for the structured design and analysis techniques commonly found in the different software design methodologies. So it appears to be the case that methodologies and languages for finite-state systems that may be fruitfully brought to bear on the analysis of event-based software requirements specifications; it is also likely to be the case that an examination of such specifications will likely suggest new approaches to finite-state analysis that have heretofore not been considered.

## 3 Tools for Finite-State Analysis

This note closes with an overview of tools that have been developed by the author and colleagues for designing and analyzing such finite-state systems. The tool set at present includes the following.

**The Concurrency Workbench** [8]. The Workbench supports numerous techniques for specifying and verifying networks of interacting finite-state processes specified in *process algebra*. In particular, two systems may be compared for equivalence as well as for relative well-definedness (useful in conjunction with a stepwise-refinement development strategy) according to several different criteria, and a model checker may be used to determine if systems enjoy specific properties formulated in a flexible temporal logic, the modal mu-calculus. Special-purpose routines also enable the location of deadlocked states and the interactive simulation of specifications. Recent versions of the system also provide flexible diagnostic-information generation facilities. The tool has been applied to the study of a variety of different communicating systems, including hardware designs, communications protocols, and process control systems by researchers from around the world.

**VTVIEW** [16, 7]. VTVIEW is a graphical editor that enables users to create structured networks of finite-state systems. The graphical language supported has a formal semantics given in two different fashions. One defines the system transitions a system may engage in, while the other takes the form of a translation into a language supported by the Workbench. The latter is implemented in the tool, thereby enabling systems created in VTVIEW to be analyzed using the Workbench. The system is built on top of X-windows and is currently being alpha-tested.

**VTSIM** [13]. VTSIM, which is also being alpha-tested, provides users with a facility for stepping through the "execution" of designs created using VTVIEW. The tool includes interactive and automatic modes as well as a breakpoint feature and replay and undo facilities.

## References

[1] J. Atlee. *Automated Analysis of Software Requirements.* PhD thesis, University of Maryland, 1992.

[2] J. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–41, January 1993.

[3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Sci. Comput. Programming*, 19:87–152, 1992.

[4] T. Bolognesi and E. Brinksma. Introduction to the iso specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[5] S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14:3–23, 1987.

[6] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[7] R. Cleaveland, S. Jain, and V. Trehan. GCCS: A graphical language for network design. In preparation, 1993.

[8] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[9] D. H. et al. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–413, April 1990.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8:231–274, 1987.

[11] K. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[12] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[13] S. Jain. VTSIM: A graphical simulator for finite-state networks. Master's thesis, North Carolina State University, 1993.

[14] N. Leveson, M. Heimdahl, M. Hildreth, H. Reese, and J. Ortega. Experiences using Statecharts for a system requirements specification. In *Proceedings of the Sixth International Workshop on Software Specification and Design*, pages 31–41, Como, Italy, October 1991. Computer Society Press.

[15] V. Roy and R. D. Simone. Auto/Autograph. In *Computer-Aided Verification '90*, pages 235–250, Piscataway, New Jersey, July 1990. American Mathematical Society.

[16] V. Trehan. VTVIEW: A graphical editor for hierarchical networks of finite-state processes. Master's thesis, North Carolina State University, 1992.

# Automated Reasoning in Software Design*

Deepak Kapur
The University at Albany
Albany, New York 12222

## 1 Introduction

We report on progress made in automated reasoning over the last decade or so. We discuss recent work in our group on automated reasoning, visualization of proofs and specification languages. Finally, we outline possible aspects of software engineering and system design, where automated reasoning research can be beneficial.

## 2 Automated Reasoning and Verification

The last decade has witnessed significant advances in automated reasoning methods and development of powerful and sophisticated automated reasoning programs for inductive reasoning, first-order reasoning, equational reasoning, higher-order reasoning, algebraic reasoning, model-checking, etc. For instance, researchers at the Argonne National Laboratory have used their OTTER program for resolution-based theorem proving to settle open mathematical and logic problems. Difficult geometry problems can be easily solved using Wu and Chou's program as well as programs I have developed using Gröbner basis and characteristic set methods. We will not attempt to survey these developments, but mention a few significant ones insofar as they relate to system and software verification.

A particular mention should be made of the impressive work going on at Computational Logic Inc. using the Boyer and Moore's theorem prover. In a special issue of *J. of Automated Reasoning* edited by J Moore, this effort called *CLI short stack* was described in a series of five papers [1]. To quote Moore, "The short stack is unique because it has been verified by computer, beginning with a simple applications program in a high-level language down through the gate-level design of a microprocessor." Papers discuss how Boyer and Moore theorem prover was used to verify a multi-tasking system, a compiler, an assembler, a linker and a micro-processor. More recently, Yuan Yu, a student of Bob Boyer, has verified machine code of many utility programs in C library on Motorola MC68020 using Boyer and Moore's prover. These programs include the GCD, Quick Sort, Binary Search and other programs[2].

Of course, there is impressive work by others also including significant advances made in using model-checkers for hardware circuits, controllers and real-time systems. And, then there is a devoted community of HOL users hacking tactics and formalizing systems.

In our research group, we have been emphasizing developing methodologies for design, specification and verification of generic components [3]. By applying formal methods to generic components, we

*Partially supported by NSF Grant Number CCR-9303394.

are likely to attain improvements in the cost-effectiveness of applying formal methods to software development, since the cost for a generic component, though high, can be amortized over its many uses. The software built from generic reusable components is likely to result in improved structure and documentation for purposes of coordinating development in large projects and simplifying maintenance and future enhancements. By abstracting not only away from implementation, but also from behavior and carefully engineering interfaces, it is possible to produce generic components that can be much more easily composed and widely usable than the specialized components usually constructed. We have been using our theorem prover *RRL* (*Rewrite Rule Laboratory*) and *Tecton* proof system for understanding requirements for automatically generating and structuring proofs for properties of generic algorithms [4]. In particular, we have investigated sorting algorithms, string matching algorithms and a partitioning algorithm.

## 2.1 Induction, Decision Procedures, Proof Structuring and Visualization

It is quite clear to those of us involved in the use of automated tools for specification analysis and verification of properties of software that proofs by induction play an important and crucial role in this application. Over the last 15 years, impressive advances have been made in automating induction theorem proving methods, and many heuristics have been developed and successfully demonstrated on difficult problems. These methods have been implemented in theorem prover such as NQTHM (Boyer-Moore theorem prover), RRL (Rewrite Rule Laboratory) and NEVER. Simple properties about recursive definitions can be automatically proved by induction; appropriate intermediate lemmas needed for these proofs can be automatically generated.

There is still a great deal of work that needs to be done to enhance the automatic capabilities of theorem provers, particularly in regards to appropriate use of known lemmas from libraries as well as automatic generation of intermediate lemmas necessary for simple proofs to go through automatically. Libraries of properties of frequently used data structures such as numbers, lists, sequences, arrays, etc., need to be integrated into the theorem provers so that it is possible to build upon other's work. In this regard, integration of decision procedures for linear arithmetic, arrays, records, and lists into a theorem prover for induction is very significant. Proofs generated using integrated decision procedures and known lemmas from libraries are compact and easier to understand, emphasizing relevant detail. Model checking methods and binary decision diagrams for handling boolean formulas may also need to be integrated for dealing with real-time software as well as for hardware systems.

In our work, we have been paying special attention to user-interface issues related to theorem provers, and visualization aspects of proof attempts [4], in particular designing structured proof representations using graphics icons and hyper-text. Much like software, successful proof attempts can be reused. Related theorems have related proofs. A proof of a related theorem can be obtained by slightly modifying inference steps used in a proof of another theorem. This is especially evident while carrying out proofs for reasoning about generic components. Proofs can be parameterized as well as generalized by identifying and abstracting common patterns of inference steps. For developing support for specifying and verifying generic components, it will be useful to develop adequate representations of generic proofs in the form of proof plans and tactics.

Most verification systems do not provide adequate tools to deal with these issues. The structure of the proofs is buried in a style of linear representations most suitable for texts. The theorems and lemmas used in proofs are not readily available with the proofs, and have to be looked up in an often large list of mostly irrelevant theorems and lemmas.

In Tecton our approach for escaping such limitations combines the use of hypertext, graphical, and tabular representations to show clearly the structure of proofs. An important advantage of hypertext technology is its use in documenting, highlighting and analyzing dependencies among formulas, lemmas and inference steps. Possible effects of a change in a definition can be easily traced in a proof management system, and for each proof which uses the definition being changed, it must be checked whether redoing the same inference steps using the modified definition would lead to a successful proof attempt. If not, old proofs must be invalidated, the status of a formula should be changed from a theorem to a conjecture, and a new proof attempt should be tried. In case of multiple proofs residing in the system, proofs invalidated because of a change in a definition must be redone or deleted.

Theorem provers are lousy in providing decent user interfaces, documentation and user guides about how to use them effectively. There is a need to develop decent user interfaces and methodologies for presenting proofs which highlight proof structures at different levels of details. Unsuccessful proof attempts can also provide useful information about specification and code. Tools need to be developed to analyze failed attempts and extract useful, relevant information from them.

## 3 How can Automated Reasoning Methods Contribute?

Despite these impressive demonstrations, formal methods are not being used as widely as they should be. We are interested in finding out why that is the case. How can formal methods be integrated in to the software development cycle, irrespective of design methodology being used? What are the obstacles? How are rigorous modeling and analysis (which appear to be an integral part of requirements and design phases) practiced? We can easily see automated reasoning tools being relevant and useful there. How can that be achieved?

If one looks into any book on any *software methodology*, there seems to be unanimity on the view that requirement specifications should be clear, complete and consistent. Yet, we are not aware of any formal tools used to ensure these properties. We are interested in finding out what methods, tools and techniques are used in practice to ensure these important properties of requirement specifications. We would like to conjecture that existing automated reasoning tools can be adapted to provide some support for checking such properties of requirement specifications.

A crucial aspect of the phase of system design is to validate that the design indeed meets the requirements specifications. The importance and significance of this validation is well-recognized but again, it is unclear how this is ensured.

Is it the case that formal methods are not popular because existing specification languages/logic and theorem provers do not provide an adequate fit with methodologies used in system design? If so, why not? We believe that structuring mechanisms used in mathematics and engineering have a great deal to offer for system design. We would like to offer as an example, the design of a large software project at IBM, namely the computer algebra system Axiom.

In the Tecton specification language for specifying generic system components, we have emphasized structuring principles and abstraction [3]. The language provides *definition, abbreviation, extension*, and *lemma* constructs, which have general mathematical descriptive power, plus a computation-specific *realization* construct. The semantics includes specification of the requirements ("legality conditions") that must be met when using each construct. We believe that the language should be able to directly support object-oriented design methodologies. We plan to discuss more about our specification language at the workshop.

## References

[1] W. Bevier, W. Hunt, J S. Moore and W. Young, Special issue on **System Verification,** *J. of Automated Reasoning,* 5 (4), 1989.

[2] R.S. Boyer and Y. Yu, "Automated correctness proofs of machine code programs for a commercial microprocessor," Proc. *Automated Deduction - CADE-11 (11th Intl. Conf. on Automated Deduction,* (ed. D. Kapur), Springer LNAI 607, Saratoga Springs, NY, 1992, 416-430.

[3] D. Kapur and D.R. Musser, *Tecton: a framework for specifying and verifying generic system components*, Rensselaer Polytechnic Institute Computer Science Technical Report 92-20, July, 1992. Invited talk at *TPCD Conf. 1992 (Theorem Provers in Circuit Design,* University of Nijmegen, The Netherlands, June 22-24, 1992.

[4] D. Kapur, D.R. Musser, and X. Nie, "The Tecton Proof System," invited talk, *Proc. of a Workshop on Formal Methods in Databases and Software Engineering,* Montreal, May 15-16, 1992 (Alagar, Lakshmanan, Sadri (eds.)), Workshop in Computing Series, Springer Verlag, 54-79.

POSITION PAPER

# Analysis of Critical Non-Functional Factors of Systems

by John Salasin and Douglas Waugh
Software Engineering Institute

Non-functional, or quality, aspects of large systems are often treated in an ad hoc manner -- even when they are critical to the system's ultimate success. It is usually difficult to defend claims about a system's reliability or performance, for example, before large portions of the system have been implemented and tested. A high reliance is placed on the skill and experience of the system's designers to make sure that these quality aspects will be present once the system is fielded, but there is little evidence to comfort the program manager that this trust is well-placed until the actual fielding takes place.

There has been a long history of efforts to define and evaluate quality of software. Much of the previous work in this area has focused on either the process used to develop the software or on the analysis and testing of "finished" software products.

We define a systematic way of integrating concern with non-functional attributes from the earliest stages of the life cycle. Through this approach, we attempt to:

1) Ensure that non-functional attributes are identified and included in the architecture/design at the earliest possible stage.
2) Provide a means for tracking required non-functional attributes to ensure satisfaction.
3) Provide quantitative measures of degree of satisfaction of non-functional attributes.

This approach was developed in support of the Ballistic Missile Defense Office (BMDO). The BMDO is developing a multi-layered, object-oriented Information Architecture for the Ballistic Missile Defense (BMD) Battle Management/Command Control Communications (BM/C3) System. Our goal in analyzing non-functional aspects is to assure that one can easily identify, measure, and track the non-functional aspects of the developing system at each stage of design, implementation, and operation.

Non-functional critical quality factors for the BM/C3 system include desired operational behavior and design characteristics relating to:

- timeliness (stringent performance requirements)

- useability (operability, understandability, ease of use, etc.)
- dependability (availability, integrity, fail-safe, trust, etc.)
- adaptability (maintainability, evolvability, etc.)

Our approach requires annotation of the architecture at the most abstract levels with non-functional requirements called obligations. A stepwise refinement process is used to refine obligations into commitments (and other obligations) in concert with the development of the layered architecture. The refinement process employs defined metrics and indicators to validate each refinement step.

To illustrate the stepwise refinement of obligations and commitments, consider the ***obligation*** *--"continued operation in case of computer hardware failure"*

This could be refined into: ***commitment a*** *--"operating system will detect hardware failures"*; ***obligation b*** *-- "backups specified for hardware (hot/cold spares)"*; and ***obligation c*** *--"automatic switching to backup processor"*.

The obligation labeled **obligation b**, then, could be refined into the ***commitment d*** *-- "satisfied by behavior specification for Initialization/Configuration function"*. And **obligation c** could result in 3 additional ***obligations*** -- (1)*"receive and process interrupt"*, (2)*"save system state"*, and (3)*"transfer control to backup"* and so on, until all obligations have been committed.

The approach also defines a set of indicators and metrics that enable quantitative and qualitative assessments of the Information Architecture's treatment of non-functional factors. These indicators/metrics fall into 3 categories:

1. existence: Has a process or rule been defined to assure satisfaction of an attribute. For example, is there a monitor function or object specified to assure task completion? are there rules regarding layering and encapsulation of exception handdling mechanisms?

2. allocation: Has the satisfaction of the desired attribute been allocated to the hardware/software infrastructure, design rules and conventions, or to the application design itself?

3. adequacy: How likely is the specified mechanism to ensure satisfaction of the desired attribute?

Examples of indicators in the 3 categories are shown in the table below. Consider, for example, the survivability factor. An indicator of satisfaction of the requirement is that objects have been defined in the architecture that monitor for task completion; An allocation indicator is the notation that the operating system will be responsible for detecting hardware failures; And the adequacy metric shows a high score for the solution chosen -- automatic switching to hot spares.

**Examples of Indicators and Metrics for Non-Functional Factors**

| FACTORS | INDICATORS and METRICS | | |
|---|---|---|---|
| | EXISTENCE | ALLOCATION | ADEQUACY |
| Testability | pre and post conditions for processes | boundary testing to be covered by test plan | High: formal behavior specs |
| Survivability | monitor objects for task completion | OS responsible for detection of hardware failures | High: auto switch to hot spares |
| Info. Validity | value range annotations on object attributes | Rule: use Ada strong typing with constraint checking | Low: rely on sensor processors to screen |
| Useability | annotations to indicate points of human interaction | Rule: control board to rule on HCI screens/messages | High: real user involved in early HCI prototyping |
| Trust | objects and processes annotated with trust levels | OS to provide priviledged interface / instructions | High: formal methods/proofs |
| Evolvability | class hierarchy provides levels of abstraction | Style rules for design and programming | Med: analysis of likely changes |

By providing a set of metrics and indicators along with a process that complements the design process in use on the project, we enable system architects to defend their architecture decisions in terms of quality of the system which will result.

We believe that this approach offers several benefits:

To the architecture team -- The early focus on non-functional aspects forces out common definitions of quality attributes. The approach provides a capability to examine alternative solutions or means for satisfying desired quality atteibutes. And it makes requirements on infrastructure and design rules and conventions explicit and visible.

To the design and development team -- The approach provides through the Information Architecture a well reasoned approach to satisfying quality requirements as well as characteristics and requirements for the hardware/software infrastructure selection and guidelines for the software design.

To the validation/verification/test team -- It provides early opportunities for V&V assessment and specific test requirements.

# Toward Practical Applications of Software Synthesis *

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304
smith@kestrel.edu

My colleagues and I at Kestrel Institute have been exploring automated tools for transforming formal specifications into efficient and correct programs. In particular, I have specialized in automating the design of algorithms. KIDS (Kestrel Interactive Development System) [3] serves as a testbed for our experiments and provides tools for performing deductive inference, algorithm design, expression simplification, finite differencing, partial evaluation, data type refinement, and other transformations. We have used KIDS to derive over 60 algorithms for a wide variety of application domains, including scheduling, combinatorial design, sorting and searching, computational geometry, pattern matching, and mathematical programming.

Formal methods need some dramatic success stories to spur the interest of industry and government (and even the U.S. academic community!). My current strategy for selling the ultimate practicality of tools to support formal methods is to synthesize high-payoff algorithms, specifically scheduling algorithms. The intent is to show that automated algorithm design tools can economically generate families of high-performance scheduling codes. To this end we are producing a scheduling synthesis workstation that combines general-purpose synthesis tools with extensive knowledge about the scheduling domain and effective programming techniques for scheduling.

Tremendous benefits arise from having good scheduling systems. Many practical scheduling problems are NP-hard, so it is likely that there is no general and efficient solution method. However the problem cannot be avoided – it *must* be solved. The intrinsic combinatorial difficulty of scheduling practically requires heuristic algorithms for solving large-scale problems – optimal schedules can only be obtained for problems involving tens or hundreds of activities. The suboptimal schedules produced by most schedulers means that time, money, and resources are wasted.

As part of the ARPA/Rome Laboratories Planning and Scheduling Initiative, we have focused on the transformational development of transportation schedulers [2]. Our approach involves several stages. The first step is to develop a formal model of the transportation scheduling domain, called a *domain theory*. Second, the constraints, objectives, and preferences of a particular scheduling problem are stated within a domain theory as a *problem specification*. Finally, an executable scheduler is produced semiautomatically by applying a sequence of *transformations* to the problem specification. The transformations embody programming knowledge about algorithms, data structures, program optimization techniques, etc. The result of the transformation process is executable code that is guaranteed to be consistent with the given problem specification. Furthermore, the resulting code can be extremely efficient.

The U.S. Transportation Command and the component service commands use a relational database scheme called a TPFDD (Time-Phased Force and Deployment Data) for specifying the

*This research was supported in part by DARPA/Rome Laboratories under Contract F30602-91-C-0043, in part by the Office of Naval Research under Grants N00014-90-J-1733 and N00014-93-C-0056, and in part by the Air Force Office of Scientific Research under Contract F49620-91-C-0073.

transportation requirements of an operation, such as the Somalia relief effort. We developed a domain theory of TPFDD scheduling defining the concepts of this problem and developed laws for reasoning about them. KIDS was used to derive and optimize a variety of global search scheduling algorithms that are generically called KTS (Kestrel Transportation Scheduler). The KTS schedulers are extremely fast and accurate (see below).

Transportation scheduling tools currently used by the U.S. government are based on models of the transportation domain that few people understand [1]. Consequently, users often do not trust that the scheduling results reflect their particular needs. Our approach tries to address this issue by making the domain model and scheduling problem explicit and clear. If a scheduling situation arises which is not treated by existing scheduling tools, the user can specify the problem and generate an situation-specific scheduler.

There are several advantages to a transformational approach to scheduling. First, there is no one scheduling problem – there are *families* of related problems in any given scheduling situation. The problems can differ in the mix of constraints to satisfy, cost objectives to minimize, and preferences to take into account. A typical problem is to schedule a given collection of activities on given resources. Another kind of problem is to find an estimate of the resources needed to bring about a desired completion date. Another kind of problem is to work backwards from a given completion date to feasible start dates for individual activities. Another kind of problem is incremental or reactive scheduling. We believe that transformation systems such as KIDS will provide the most economical means for generating such families of schedulers. We have observed a great deal of reuse of concepts and laws from the underlying domain theory and of the programming knowledge in the transformations.

A second advantage is the reuse of best-practice programming knowledge. The systematic development of global search algorithms has helped us exploit problem structure in ways that other projects sometimes overlook. The surprising efficiency of KTS stems from two sources. First, the derived pruning and propagation tests are surprisingly strong. The stronger the test, the smaller the size of the runtime search tree. In fact, on many of the TPFDD problems we've tried so far, KTS finds a complete feasible schedule without backtracking! The pruning and propagation tests are derived as necessary conditions on feasibility, but for this problem they are so strong as to be virtually sufficient conditions. The second reason for KTS' efficiency is the specialized representation of the problem constraints and the development of specialized and highly optimized constraint operations. The result is that KTS explores the runtime search tree at a rate of several hundred thousand nodes per second, almost all of which are quickly eliminated.

The chart in Figure 1 lists 4 TPFDD problems, and for each problem (1) the number of TPFDD lines (each requirement line contains up to several hundred fields), (2) the number of individual movement requirements obtained from the TPFDD line (each line can specify several individual movements requirements), (3) the number of movement requirements obtained after splitting (some requirements are too large to fit on a single aircraft or ship so they must be split), (4) the cpu time to generate a complete schedule, and (5) time spent per scheduled movement. Similar results were obtained for sea movements.

We have compared these results with other schedulers. The OPIS project at CMU takes a similar declarative approach to modeling scheduling as a constraint satisfaction problem. OPIS requires about 30 minutes to solve the CDART data and it does not find a complete feasible schedule (some latest arrival dates are relaxed). KTS finds a complete feasible solution in 0.4 seconds – a factor of 4500 faster.

We have also compared these results with the PFE (Prototype Feasibility Estimator), which is a CommonLisp re-implementation of a military feasibility estimator called TFE (Transportation Feasibility Estimator). Including preprocessing time, PFE takes about 206 seconds on the

| Data Sets (Air only) | # of input TPFDD records | # of individual movements | # of scheduled movements after splitting | Solution time | Msec per scheduled movement |
|---|---|---|---|---|---|
| CDART | | 403 | 539 | 0.4 sec | 1.0 |
| CSRT01 | 624 | 1271 | 3120 | 8.3 sec | 2.6 |
| 090TP/PFE | 4471 | 6160 | 8085 | 27 sec | 3.3 |
| 9002T Borneo | 9480 | 12370 | 15460 | 71 sec | 4.6 |

Figure 1: Scheduling Statistics

090TP/PFE data to schedule the sea movements and estimate the schedulability of the air movements. KTS is 78% faster, taking 43 seconds to produce a detailed feasible schedule for *both* air and sea movements. Furthermore the KTS schedule produces 75% less delay in the sea movements and provides a far more accurate measure of the number of planes required for the air movements.

From [1] it appears that KTS is hundreds of time faster than active duty transportation schedulers FLOGEN (for air movements) and SEACOP (for sea movements). We plan to visit Scott AFB later this year to compare KTS with the JFAST system at UTC (US Transportation Command) and the ADANS system at AMC (Airlift Mobility Command).

To conclude, it appears that there is an opportunity to demonstrate that formal software development tools could fill a real need for high-performance schedulers. To get around the problem that schedulers are usually embedded within a larger system, we develop "plug-compatible" interfaces to our derived schedulers. In this manner we have delivered KTS into the Common Prototyping Environment at Rome Labs. The idea is to allow the substitution of our schedulers for existing schedulers and to perform comparative experiments. This seems the most tenable way to allow formally developed code to be useful in legacy systems.

## References

[1] John Schank. et. al. *A Review of Strategic Mobility Models and Analysis*. Rand Corporation, Santa Monica, CA, 1991.

[2] Smith, D. R. Transformational approach to transportation scheduling. In *Proceedings of the Eighth Knowledge-Based Software Engineering Conference* (Chicago, IL, September 1993).

[3] Smith, D. R. KIDS - a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024-1043.

## Summary and Conclusions
## Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development

Valdis Berzins

### 1. Summary of the Presentations and Discussions

The main theme of the presentations and discussions at the workshop was how to provide computer support for combining changes to software systems, which is the software merging problem. Secondary themes at the workshop were the supporting technologies that are relevant to solving the software merging problem, and other aspects of computer support for software changes.

Software merging has subproblems that include decomposing software into independent parts, recombining parts, and choosing the parts of the different versions to recombine. A special case of the software merging problem is one where all of the changes are pure extensions to the behavior, rather than modifications. The presentations by Mili, Sterling and Huang explored this special case in different contexts. A result of the discussions was that all of these efforts appeared to be aimed at constructing least upper bounds in different lattices. A common point of Mili's work and Sterling's was the desire to allow different people to work on different aspects of a system and then to put the parts together with some guarantees of consistency if the process succeeded. Huang was more concerned with decomposing a program into parts, simplifying the parts, and recombining them, so that consistency was not an issue in that context. The unrestricted problem, including changes that could modify or retract the behavior of the system as well as extend it, was considered by Dampier (for prototypes with real-time constraints) and Feather (for requirements specifications). Dampier's approach was formal and relied on a behavior invariance theorem for the prototyping language for its correctness. Feather's approach was informal, partially because of the social issues that interact with the early stages of requirements formulation. The elaboration structure of the requirements was used mainly as a vehicle for simplifying explanations of the requirements to the clients, and relied on human thinking to detect interactions between "almost independent" elaborations.

One of the supporting technologies for software merging is program slicing, which can be used for the decomposition aspect of the problem. The presentation by Dampier showed how slicing could be applied to combining unrestricted changes to prototypes with real-time constraints, using an augmented dataflow representation that has a formal semantics. Other applications of program slicing include debugging, simplification, and change impact analysis (which uses forward slicing instead of backward slicing), and can be used to provide guidelines to concurrent updaters to prevent conflicts, as pointed out by Agrawal and others.

Another supporting technology is meaning-preserving transformations. Such transformations can be used as a normalizing procedure that enables sound recognition of some classes of behavioral equivalence for programs, which is another subproblem of software merging. Other applications of meaning-preserving transformations that were explored at the workshop include program simplification to aid understanding (Huang) and restructuring to make subsequent semantic changes easier to realize, both in terms of effort and accuracy (Griswold). Anecdotal evidence from the participants suggested that restructuring is often a large part of the effort to realize a "difficult" change in software behavior. Sterling indicated that in some cases restructuring was needed to avoid unification problems with merging (i.e. in joins of PROLOG programs).

Transformations that make refinements were also discussed at the workshop, and it was suggested that monotonic and correctness conserving transformations were better names than meaning preserving transformations, because the transformations could add information that consistently extends the meaning of the software artifact (program, specification, etc.).

The problem of how to realize evolution for systems that must provide continuous service, was explored by Mittermeir. The main point of the presentation was that the interface of a module must be decoupled from the realization of the operations via an intermediate level that can be updated. Some of the questions about the details of this intermediate level had to be deferred because the project related to these aspects is still in its early stages.

The problem of automatically finding test data that would drive a program down a specified path was explored by Antoy. This problem is similar to one of the subproblems that arises in change merging, which is to decide if it is possible for two path conditions to be satisfied by the same input data. The approach used was to apply the narrowing algorithm for solving symbolic equations over abstract data types. The formal method was very clean, and had associated completeness results and optimal efficiency results, in the sense that the method explored only those paths in the derivation that could not be avoided without discarding some solutions. The method worked very nicely for a variety of small problems. It was pointed out that the completeness result seemed to imply that the method could not scale up, because the problem that was being solved had at least (single, double, or triple) exponential complexity, depending on what restrictions were imposed.

## 2. Conclusions of the Workshop

The opinions summarized below emerged from the workshop and received more or less general agreement from the participants.

(1) Since all nontrivial behavioral properties of programs are undecidable, interesting software development tools are necessarily incomplete. The general feeling was that this is not a problem, provided that the tools treat cases that were too difficult to decide as failures. Such tool incompleteness was considered to be a useful diagnostic for inappropriate designs - if the tools could not understand the design, that was an indication that it was too complex and too hard, and should be corrected to make it maintainable. The real challenge is "how much can we do automatically?".

(2) A related difficulty is demonstrating the usefulness of a tool that does something better than we can do manually or does something that is too difficult to be done manually. Explanations of small portions of the tools can help provide users with an intuitive "happiness" with the tool, and empirical tests can provide evidence to support the feeling that "I now believe in the tool". Another option is to provide methods for tracing the decision processes of the tools, and different "verbosity levels", so that users have some rational options when the tool makes a non-intuitive decision (that may nevertheless be correct). A related issue is integrating computer support for software evolution with reliability engineering and performance engineering applied to the tools because we may not completely trust the tools. This is entirely rational because the tools themselves are complex software systems that are attempting to solve problems that have not been previously automated. Consequently there may be subtle requirements errors as well as implementation errors induced by complexity and confusion.

(3) Perhaps facilitating slicing and merging should be another guideline for organizing specifications, designs, and programs. This is a different dimension of modularity: "software design for slicability and mergability". This may not be easy because change

merging does not distribute over functional composition (as expressed by the inequalities in the opening remark slides, see [1].) This means that changes to different parts of a dataflow decomposition are not in general independent. The practical goal is to be able to make reliable local changes without having to understand or analyze the whole program. One approach is to assign slices rather than modules as work units, but some problems are that slices can be rather large, and different slices can overlap.

(4) The question "what is a change?" needs more attention. There are different answers at different levels. Lee White (Case Western) has characterized types of changes in the context of regresssion testing. Mutation testing provides another view, which is structural rather than behavioral. The functional view explored by Berzins and Reps is based on a decomposition of functions into the part that is the same in both versions, the part that is added by the change, and the part that is removed by the change. At the specification and requirements level we can talk about constraints or goals that are preserved, added or removed. There is probably a low consensus on what a change is and what the different levels of change mean. Perhaps a taxonomy of changes is in order. Reasoning about the code is not enough, we need to consider different views, including design rationale. The higher level information can help to resolve conflicts, but mapping high level modifications back to the code level is a tough problem. Decisions to tailor the requirements can be motivated by impact on the context of the system, feasibility, cost, performance, etc. Changes to data, both at the schema level (classes and subtypes) and at the instance level, and the interaction with consistency constraints should be explored. Another issue is the relation to specification languages and inheritance mechanisms.

(5) Remember to keep the highest levels goals in mind. Techniques are not the core of the workshop, but rather creating reliable and correct software at low cost, as noted in the list of goals for the workshop (page 6).

(6) If we make software more easily modifiable, it is not clear if the costs associated with modification will really go down, or if more modifications will be made because it is easier to do. Hopefully a positive answer will lie in the extended lifetime of maintainable systems and less discarding and rebuilding of systems that are almost correct but cannot be changed because no one can understand them any more or because the changes would require manual fixes to most of the code.

## 3. Acknowledgement

We would like to thank all of the participants of the 1993 Monterey workshop for their active participation and insights, which enabled us all to create a successful workshop.

## References

1. V. Berzins, "Software Merge: Semantics of Combining Changes to Programs", Technical Report NPS 52-91-4, Computer Science Department, Naval Postgraduate School, 1990. Revised for ACM Trans. Prog. Lang and Systems.