Strategic Directions in Software Engineering and Programming Languages

Carl Gunter
University of Pennsylvania
and
John Mitchell
Stanford University
and
David Notkin
University of Washington

ACM Computing Surveys Vol. 28, No. 4 (Dec. 1996), Pages 727-737. Association for Computing Machinery, Inc., 1515 Broadway, New York, NY 10036, USA Tel: (212) 555-1212; Fax: (212) 555-2000

The techniques and tools that will be needed for efficient and accurate system development, in the face of increasing system complexity and declining life cycles, require deep, new and exciting scientific and engineering research. The challenges demand the best efforts of both the programming language and the software engineering research communities, balancing the beauty of programming language and compiler theory with the messy, ugly, often not-entirely-technical problems that arise in building and evolving real software. The synergy between software engineering and programming languages and compilers isn't merely desirable—it's essential. This report enumerates a set of four areas that demand additional research immediately and over at least the next decade.

Introduction

Software is *the* critical advance of the computer revolution, which is based on the ability of a general-purpose machine to carry out a specific task under the control of special-purpose software. While the wide applicability of stock hardware has driven demand up and manufacturing cost per unit dramatically down, the same trends have led to increasing demand for ever more complex software for an astounding variety of uses: computer-aided education, improved business processes, what-if financial planning and analysis, new forms of entertainment, fly-by-wire avionics systems, and innumerable others both known and as-yet unknown.

Producing quality software systems at reasonable cost is an increasingly serious challenge due to ever-growing economic and societal demands. Moreover, software that cannot be modified, updated or extended becomes useless, due to changes in its technological, social, and economic context. Successful evolution of software systems is a severe technical and managerial challenge that is only exacerbated by the ever broadening kinds of software that must be produced and changed.

The techniques and tools that will be needed for efficient and accurate system development, in the face of increasing system complexity and declining life cycles, require deep, new and exciting scientific and engineering research. The challenges are staggeringly hard, demanding the best efforts of both the programming language and the software engineering research communities. Specifically, we must use impressive advances in programming languages, compiler technology and software engineering, balancing the beauty of programming language and compiler theory with the messy, ugly, often not-entirely-technical problems that arise in building and evolving real software. The synergy between software engineering and programming languages and compilers isn't merely desirable—it's essential. With hard work, open minds, and appropriate resources, these two communities can apply existing results to make significant progress in bringing science and engineering together to address the challenges of building and evolving diverse software systems.

To clarify the potential for synergy between the communities, consider two examples.

—Domain-specific languages, developed specifically to implement a family of related software systems, have proven effective in a wide variety of applications. While some theoretical frameworks may not meet the challenge for general-purpose programming languages, corollaries or special cases of these theories may well apply to some domain-specific languages. An example is the call-by-name lambda calculus. Although there is a huge body of theory about it, call-by-name has been largely rejected as a modern feature for general-purpose programming languages. On the other hand, it provides an elegant and practical for a domain-specific language for software configuration [Abadi et al. 1996].

—Analysis of source code arises in both communities. In the design and implementation of programming languages, analysis is critical to compiler optimization and parallelization. In software engineering, analysis may be used for program-understanding tools such as program slicers or for producing program databases that are useful for tasks such as reverse engineering Although these areas have a great deal in common, the analyses that are performed have significant differences. In particular, compiler analyses must generally be conservative, while software engineering analyses may, in some cases, be non-conservative. Pursuing a general theory of analysis that captures the essence of the commonalities in the context of different requirements would be valuable and is clearly feasible.

In contrast to many areas in computer science and engineering, the strategic goal of these two communities is not solely, nor even primarily, to make software systems bigger, faster, and cheaper, but rather to make it possible to build and evolve more, new software systems. The main body of this report is devoted to four specific research areas that have been identified for their potential as the result of working-group activities. These four areas are not intended to be a complete or exclusive description of all valuable opportunities. However, they do illustrate the potential for synergy between software engineering and programming language research activities, with the goal of solving timely and pressing problems effectively. An appendix summarizes the series of meetings that led to this report. A planned extended version of the report also contains programmatic recommendations.

The four areas are

(1) the design of languages for programming over the internet and the World Wide Web;

(2) the design of domain-specific programming languages;

(3) techniques for relating programming languages to specification languages; and

(4) the use of programming language compiler analysis ideas to support software engineering in areas beyond compiler optimization.

We consider them in four sections.

1. PROGRAMMING THE WEB

An area of explosive growth in computing is that of the internet and the World Wide Web (WWW). Computing over the internet and WWW provide unique challenges whose solutions will involve the development of systems beyond the capabilities of present day programming languages and software engineering tools and practices.

Key properties needed to exploit the Web will include:

—ensuring proper security,

—allowing distributed software development,

—supporting mobility (remote agents), and

—ensuring global requirements (properties of the network as a whole).

One promising approach to producing systems that have these properties is to produce specialized programming languages. Applying best practices for programming language design early and often is desirable in developing such languages, since several of them are likely to be enshrined as widely-used standards. Early use of such best practices should be possible since the issues in this area can often be addressed by well-developed ideas from programming languages that have not yet found their ideal applications.

Security is a global and fragile property whose integrity cannot be entrusted even to conscientious users and programmers. Nor would users and programmers want to be responsible for manually maintaining subtle security requirements. Therefore, a number of automated and semi-automated techniques will have to be developed to enforce various levels of security while guaranteeing reasonable usability. These techniques will address issues such as: encryption; security protocols (to support mobility, for instance); policies (authentication and establishing permission); and syntactic, static checking, and analysis of security properties. (Security, like many other topics addressed in this report, are also clearly the subject of ongoing research in a number of other areas in computer science and engineering; the areas need to cooperate as appropriate to ensure progress without undue overlap.)

The internet enables and encourages programming with off-the-net *distributed software* components. This possibility raises a variety of software engineering and programming language issues:

—integration and testing of distributed software,

—stronger modularity requirements,

—support for first-class modules and interfaces (dynamic inlining),

—configuration management (both pre- and post-deployment), and

—distribution, evolution, and maintenance of web software.

For instance, with respect to the problem of configuration management, could changes in a very widely-used web component result in a massive automatic world wide recompilation of dependent web-based software?

To deal with the practicalities of distributed computation it will become increasingly desirable for net computations to be become *mobile*, sending an electronic "agent" to a remote host to do the bidding of the sender. Conversely, users will need to receive agents able to interface them intelligently with remote hosts. This form of computing is largely new and raises some interesting technical issues:

—Resource discovery and adaptation: How will an agent learn what resources are available on a host and adapt to what it learns?

—Resource control: how will a host control the degree to which an agent uses its resources? How will an agent be guaranteed adequate resources?

—Locality: How will an agent be able to tell where it is and "how far" it is from certain resources so as to optimize its movements and operation?

—Security: how will a host protect itself from illegitimate "inspection" by other agents?

One issue currently under investigation with Java—a programming language whose design provides security support for mobility—is the right balance of guarantees based on static/run-time checks and encryption. There is an opportunity and need for applying ideas from programming language theory to the formulation and proof of desired properties.

Possibly one of the most challenging areas of investigation will be the formulation of *global requirements* for networks like these. As networks become more increasingly important, it will be essential to predict their behaviors and possibly set up safeguards against two classes of problems:

—Attacks intended to deny service or provide wrong results, and

—"Bad" cooperative behavior.

The first of these categories is related to the problem of "information warfare" in which network havoc is part of a malicious strategy to harm the users and agents of a network. If a large part of commerce in the U.S. comes to be based on the internet, for instance, then an outage could result in significant economic inconvenience, losses, or even dislocation. The second category applies to situations like electronic trading on the stock market where the interaction of a collection of program traders presumably guided by the invisible hand of the free market could go unexpectedly off-track as it did in October of 1987. Sample question: what is the requirement that the program trading collars imposed on the NYSE after the '87 crash were intended to enforce and was this aim achieved?

Little formal work has been done on expressing requirements in these categories, despite their overwhelming importance. It seems at least plausible that parts of the solution will be derived from ideas about programming languages, where there is a great deal of experience with guaranteeing properties by static and dynamic analysis. Moreover, the development of formal means of expressing requirements in this domain may be especially desirable.

2. PROGRAMMING THE DOMAIN

Domain-specific languages are intended to allow domain engineers to develop families of applications that are easily specified, highly evolvable, and largely automated. The objective is to move a significant amount of the software development and maintenance burden from conventional software engineers and programmers to people who are experts in the domain of the applications. There are two primary reasons for this. First and foremost, the experts in the areas have, by definition, direct and deep knowledge of the area, which reduces (and might eliminate, in some cases) the problem of having non-experts in the domain make inappropriate decisions about the domain. Second, this approach has the fundamental benefit of allowing a larger number of software systems to be created. As a high-level example of domain-specific programming, millions of non-programmers write and maintain spreadsheets to perform multitudinous tasks; imagine if each spreadsheet had to be written by a programmer with four years of college education!

Perhaps the key challenge in this area is to develop the tooling that allows such domain specific languages, and the run-time resources on which that family of applications relies, to be created by domain experts.

Previous domain specific approaches have focused on the run-time resources required by applications in that area. These resources have been organized into libraries and/or abstract data types. This approach augments those efforts by adding a linguistic notation that enables a concise specification of the application, generation to produce the application that utilizes those run-time resources, and domain specific analysis (such as test-case generators) and optimization. Advances in software engineering's understanding of the issues in domain-specific languages and in the techniques to address them, along with programming language and compiler technology that is applicable to the topic, must be brought to bear.

Specific research problems we identified as requiring additional effort include:

—Reducing the cost and compiler expertise required to produce a domain specific language and product line.

—Domain specific analysis and optimizations.

—Formalizing the informal notations found in a domain.

—Integrating multiple domain specific specifications into a single system specification.

—Incorporating best programming languages practices into domain specific languages.

—Increasing evolvability through increasingly declarative specifications.

—Evolving the language and compiler/generator.

3. PROGRAMMING THE SPECIFICATION

A recent event that underscores the importance of further work on specification was the detonation of the new Euro-rocket Ariane 5 (with commercial payload aboard) earlier this year. A press release after investigation includes the following passage (emphasis added):

> The failure of Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift-off). This loss of information was *due to specification and design errors in the software* of the inertial reference system.[1]

While catastrophic failures as the result of software deficiencies have been with us for decades, there appears to be a rising awareness of the costs of software errors and a growing body of scientific and engineering work aimed at reducing the likelihood of such errors. For all of these reasons, we believe that the next decade will see far more widespread acceptance of formal and semi-formal methods in software design, development, testing and modification. Since space and military projects often use some form of requirements and design specification, we expect the changes to be most dramatic in the commercial sector. However, as with other developments in computer science and engineering, if seems very likely that wider acceptance in the commercial sector will bring advances that will benefit software design for all applications.

There are three factors driving increased interest and opportunity in software specification:

—an increasing need for precise description of what software is intended to do,

—improvements in technology for making use of formal specifications, and

—a changing focus from full system correctness to specification and validation of simpler properties of primary importance, such as security or absence of deadlock.

For instance, the software requirements for the A-7E aircraft had to be culled from a bookcase full of documentation before they were reduced to a collection of tables [Alspaugh et al. 1992] describing system state transitions. Many of the industrial uses of specification techniques focus primarily on formal specifications as a descriptive tool to achieve greater precision [Craigen et al. 1993a; Craigen et al. 1993b; Wing 1990]. Technology for writing and manipulating specifications is being provided by tools that allow more to be done with specifications and require less knowledge on the part of users. In particular, larger systems can be usefully treated by focusing on key properties such as Byzantine agreement [Gong et al. 1995] or type soundness [VanInwegen 1996].

While specification languages and automated reasoning techniques are central to the use of formal methods, programming languages are also critical. Since software systems are expressed in programming languages, any formal analysis of a software

[1]This is a quote from `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/press33.html`.

system must rely on a precise understanding of the semantics of the programming language. For example, a proof that a program does not read or write memory locations that are not allocated to it is only as good as the soundness of the logic in which it is carried out. Another aspect of programming language work is that if certain program invariants can be guaranteed by the language, then there is less need for additional tools to detect violation of the associated program properties. It is certainly the experience of all researchers in software specification, verification and testing that the presence or absence of certain programming language constructs can radically alter the difficulty of the task. (Perhaps the simplest and most graphic illustration is given in the theoretical study [Clarke et al. 1983].)

Some specific challenges for specification languages are:

—Expand the set of critically important program properties that may be expressed in specification languages. For example, dependency between modules or processes, security, safety, performance and resource use, and obligations to initialize or deallocate memory locations.

—Develop techniques for maintaining the relation between specification and software system as either or both evolve.

—Improve current experimental techniques for developing test suites based on specifications.

—Develop methods for combining the strengths of different methods for analyzing software systems. For example, the result of type checking or other form of static analysis could be used to assist a theorem prover, and when a theorem prover fails to establish a proof obligation, facts about how the prover failed could be used to generate test suites.

Although requirements and design methodologies using formal and semi-formal specifications are widely used in some circles, there are many programmers and software development organizations that either do not use such approaches, or make only minimal use. Wider success for specifications will come from providing tangible benefits, not just the potential for verification. These include

—test oracles and test generation,

—configuration management,

—simulation and modeling, and

—resource budgeting (for quality of service).

It is only by providing an incremental adoption path for existing organizations that the scientific and engineering progress in the formulation use of software specifications will achieve truly mainstream acceptance.

4. ANALYSIS AND MANIPULATION: BEYOND CODING

Many advances in programming language and compiler research can be viewed as an effort to create mechanisms to raise the level of discourse at which software developers operate. It is important to recognize that this is also a goal of much software engineering research—by raising the level of discourse, one can provide increased leverage for solving software engineering problems.

One of the ways to provide increased leverage is via program manipulation tools that aid programmers in creating new systems and in understanding, enhancing, debugging, testing, and reusing existing systems. In most cases, such tools require that certain kinds of program analyses be carried out to support the operations provided. In this section, we discuss three areas where additional research on program analysis and program manipulation can have impact:

(1) evolution [Parnas 1994; Lehman 1980] and reengineering [Chikofsky and Cross 1990],

(2) dealing with issues of binding time, and

(3) whole-program analysis.

During its lifetime, a system may undergo many alterations in response to changes in hardware platforms, software platforms, and user requirements. Today, the *evolution* of software to respond to such changing factors can only be carried out with considerable expense and risk. A case in point is the "year 2000 problem" [Hayes 1995], which requires a huge amount of software (both code and data) to be altered from their present convention of using just two digits to represent (twentieth century) dates.

The year 2000 problem is particularly pernicious, both because of the deep-seated nature of the problem and because of the (nearly) synchronous world wide deadline for addressing it. It serves to draw attention to the magnitude of the software evolution problem, but we wish to stress that this sort of problem is an everyday occurrence. The "background activity" having to do with evolution is as least as great as the year 2000 problem, and software evolution problems will not disappear in the year 2000.

Research on software evolution and reengineering should focus on ways to supply answers to such questions as: What are the invariants in a piece of software? How can we avoid violating invariants that should be preserved by a change? How can we predict the impact of a change with confidence before making it? The research challenge is to build a corpus of techniques for problems such as recognizing abstractions, identifying patterns, and determining what elements are responsible for what behavior. For example, in the year 2000 problem, these sorts of techniques would be used to answer such questions as: What is the date abstraction in this piece of code? What code is manipulating dates (and in what format)? What code is dependent on a two-digit format for dates? What data in external files represents dates?

A fundamental problem that we encounter as we try to gain leverage on software evolution and reengineering issues is that software systems are highly "multidimensional", but the different facets of a system usually have an overlaid or interleaved structure. This presents a major problem for the research community because language mechanisms are generally good for things that are contiguous or "orthogonal"; however, it is difficult to get a handle on elements that cut across the natural spatial boundaries of a system.

To some extent this can be addressed as a representation issue. For example, at the level of intermediate representations of programs, the difference between control flow graphs (a.k.a. flow charts) and program dependence graphs illustrates how different representation choices can allow elements that are non-contiguous in one representation to be contiguous in another representation.

At a much higher level, the design patterns work [Gamma et al. 1994] has raised the awareness of many practicing software developers to the issue of interleaved facets. It has been particularly beneficial for providing a vocabulary for discussing these issues. A possible opportunity for the research community is to migrate the design patterns work to language mechanisms and tools. However, some researchers have taken a different tack and shown that design pattern issues can already be addressed using the advanced features of existing languages, such as Gopher. Both avenues deserve to be pursued further.

Many issues in the development of a software system can be viewed as issues of *binding-time* commitments (or "staging" commitments). Binding-time commitments made at different points in the development of a system lead to many of the problems in software evolution, the issue being: "What do you do when your previous binding-time commitments need to be changed?" Examples of problems that arise when binding-time commitments are revisited include:

—Making systematic representation changes. (The "year 2000 problem" is an example of a change that involves a systematic representation change.)

—Migrating the position of integrity checks (between caller and callee for instance) [Scherlis 1994].

—Responding to the evolution of an API (for an API over which you have no control).

It is our thesis that viewing software engineering issues in terms of binding-time commitments is a promising way to formalize problems so as to make them accessible to the programming languages community.

Another example of a binding-time change is the migration of a system to a new implementation language. This can be addressed to a limited extent via linguistic mechanisms—e.g., by making C++ essentially upwards compatible with C. However, this is clearly only a partial solution: Even though C++ is essentially upwards compatible with C, there are still interesting issues that arise in such a conversion process, in particular, how to discover places in the code where the improved features of C++ can be exploited (such as the ability to have C++ templates). The research issue here is to devise analyses and tools that can help with the process of language migration. (Similar problems also arise with other sorts of platform migration situations, e.g., moving ordinary X applications to Ole/OpenDoc, etc.)

A third area where it appears that binding-time notions may be able to lead to advances in software-development environments is in replacement of the traditional compile-link-load cycle. The question is: "Is it possible to replace the compile-link-load paradigm by exploiting the ability to perform static specialization at a variety of points in time (not just at compilation time)." This can be supported either via linguistic mechanisms (e.g., run-time code generation mechanisms, as found in C, for instance) or by tools such as partial evaluators (e.g. Similix, Schism, CMix, etc.)

As pointed out earlier, research on program analysis has been carried out in both the programming languages and compilers and the software engineering research communities. This is clearly an area where technical advances in one community can have impact on the other. In both communities, one of the themes of the last several years has been the interest in *whole-program analysis* [Chambers et al. 1995; Atkinson and Griswold 1996], which is closely related to interprocedural analysis [Horwitz et al. 1990; Callahan and Kennedy 1988]. In the compiler community whole-program analysis propagates context information to procedures and call sites, and permits a better job of optimization to be performed. In the software engineering community, whole-program analysis permits more useful information to be reported by program-understanding tools.

Some of the challenges that we foresee in this area are as follows:

(1) *Heterogeneity.* How can analysis of multi-lingual systems be supported? Here it would also be useful to understand the range of applications in which the results of analysis are used so that researchers can tailor their efforts to techniques for gathering the most useful sorts of information.

(2) *Computational tradeoffs.* When systems are modified, facts that have been gathered via static analysis may no longer be valid. One issue concerns the tradeoffs between discarding old information and recomputing new information from scratch versus trying to update the old information incrementally in response to modifications. A third possibility is to compute information selectively (e.g., for innermost loops, in response to specific user queries, etc.) using demand-driven program-analysis algorithms. A critical aspect of these challenge is to distinguish the user of the extracted information: compilers have one need (generally, they must use conservative information to ensure that they produce translations that are consistent with the source); human programmers and software engineers may have other requirements that may permit other techniques (such as lexical ones) to be used effectively [Murphy and Notkin 1995].

(3) *Understanding related program analyses.* The tradeoffs among different, but related, program-analysis problems are not well understood. This is true at the algorithmic level, where many whole-program-analysis algorithms run in time cubic in the size of the program, in the worst case. In some situations (e.g., pointer analysis), there are near-linear-time algorithms for related problems that provide safe but less precise solutions. It would be desirable to understand the space of tradeoffs better. But this is also true in terms of the needs of the users (again, for instance, compilers vs. humans). Empirically, different design spaces are inhabited by tools for even such basic analyses as call graph extraction [Murphy et al. 1996]; understanding the relationship among the analyses for both programming languages and for software engineering is critical.

(4) *Exploiting new kinds of analysis.* There are new analysis paradigms (e.g., model checking) and new algorithmic paradigms (e.g., demand analysis) that deserve to be explored further.

Conclusions

Our report has two key points. First, it enumerates a set of four areas that demand additional research immediately and over at least the next decade. Second, it establishes that it is dangerous for the software engineering and the programming languages and compilers communities to work on these problems alone. Although not every individual researcher need try to bridge the gap across the communities, if none do then much research will be duplicated and progress in the areas will be slower than is necessary or preferable. This is more important than the specific areas that we have identified, because if we have learned anything from the past in computer science and engineering, our vision is imperfect and new areas will arise that we simply cannot predict today.

APPENDIX

History

This report is a product of a series of meetings intended to promote interaction between researchers in software engineering and programming languages.

SEPL 1. September 6-7, 1995 at Stanford.

SEPL 2. June 12-13, 1996 at MIT.

SEPL 3. June 14-15, 1996 at MIT as a working group of the ACM/CRA meeting **Strategic Directions in Computing Research**.

Participants in the SEPL 2 and 3 meetings were invited to provide position papers, which can be found through the web page for this report. The organizers (and authors of this report) gratefully acknowledge travel and moral support from ARO, DARPA and NSF.

This document was planned and written by the participants in the SEPL 3 meeting.

Related meetings include the workshops on Formal Methods in Software Engineering (sponsored on a yearly basis since 1991 by ARO), the workshop on *Future Directions in Programming Languages and Compilers* that took place in January of 1993 sponsored by NSF, and several workshops held at Dagstuhl in the past few years.

The attendees at the September 6-7 meeting at Stanford were: Alex Aiken, Univ. California Berkeley; Craig Chambers, Univ. Washington; Helen Gill, NSF; Allen Goldberg, Kestrel; Sam Kamin, Univ. Illinois; Bob Kessler, Univ. Utah; Gregor Kiczales, Xerox Parc; Richard Kieburtz, NSF; John Launchbury, OGI; Peter Lee, CMU; José Meseguer, SRI; John Mitchell, Stanford; Melody Moore, GA Tech; David Notkin, Univ. Washington; Tom Reps, Univ. Wisconsin; John Salasin, ARPA; Vivek Sarkar, IBM; Bill Scherlis, CMU; Carolyn Talcott, Stanford; Dave Wile , ISI.

The attendees at the June 12-13 meeting at MIT were: Gregory Abowd, Georgia Tech; Frank Anger, NSF; Bob Balzer, USC/ISI; Luca Cardelli, DEC SRC; Craig Chambers, Univ. Washington; Lori Clarke, Univ. Massachusetts Amherst; Dave Dampier, Army; Helen Gill, NSF; Carl Gunter, Univ. Pennsylvania; Bill Griswold, Univ. California, San Diego; Bob Harper, CMU; Paul Hudak, Yale U; Sam Kamin, Univ. Illinois Urbana-Champagne; Gregor Kiczales, Xerox PARC; Dick Kieburtz, NSF; John Launchbury, Oregon Graduate Institute; Insup Lee, Univ. Pennsylvania; Peter Lee, CMU; Karl Lieberherr, Northeastern Univ.; Dale Miller, Univ. Pennsylvania; John Mitchell, Stanford Univ.; Melody Moore, Georgia Tech; David Notkin, Univ. Washington; Jens Palsberg, MIT; J. Chris Ramming, ATT;

Tom Reps, Univ. Wisconsin Madison; Barbara Ryder, Rutgers Univ.; John Salasin, DARPA; Bill Scherlis, CMU; Val Tannen, Univ. Pennsylvania; Jack Wileden, Univ. Massachusetts Amherst; Alex Wolf, Univ. Colorado Boulder; Michael Young, Purdue Univ..

The attendees of the June 13-14 working group of the ACM/CRA meeting, **Strategic Directions in Computing Research,** were: Bob Balzer, USC/ISI; Carl Gunter, Univ. Pennsylvania; Bill Griswold, Univ. California, San Diego; Sam Kamin, Univ. Illinois Urbana-Champagne; John Mitchell, Stanford Univ.; David Notkin, Univ. Washington; Bill Scherlis, CMU; Daniel Weise, Microsoft.

REFERENCES

Abadi, M., Lampson, B., and Lévy, J.-J. 1996. Analysis and caching of dependencies. In R. K. Dybvig Ed., *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming* (1996).

Alspaugh, T. A., Faulk, S. R., Britoon, K., Parker, R. A., Parnas, D. L., and Shore, J. E. 1992. Software requirements for the A-7E aircraft. Technical Report NRL/FR/5530–92-9194 (31 August), Naval Research Laboratory, Washington, DC 20375-5320.

Atkinson, D. C. and Griswold, W. G. 1996. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering* (March 1996).

Callahan, D. and Kennedy, K. 1988. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing. 5*, 5 (Oct), 517–50.

Chambers, C., Dean, J., and Grove, D. 1995. A framework for selective recompilation in the presence of complex intermodule dependencies. In *17th International Conference on Software Engineering (IEEE Cat* (April 1995), pp. 221–30.

Chikofsky, E. and Cross, J. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE Software.*

Clarke, E. M., German, S. M., and Halpern, J. Y. 1983. On effective axiomatizations of Hoare logics. *Journal of the ACM 30*, 612–636.

Craigen, D. H., Gerhart, S. L., and Ralston, T. J. 1993a. An international survey of industrial applications of formal methods, Volume 1—Purpose, approach, analysis, and conclusions. Technical Report NRL/FR/5546–93-9581 (30 September), Naval Research Laboratory, Washington, DC 20375-5320.

Craigen, D. H., Gerhart, S. L., and Ralston, T. J. 1993b. An international survey of industrial applications of formal methods, Volume 2—Case studies. Technical Report NRL/FR/5546–93-9582 (30 September), Naval Research Laboratory, Washington, DC 20375-5320.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley.

Gong, L., Lincoln, P., and Rushby, J. 1995. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In *Dependable Computing for Critical Applications—5* (Champaign, IL, Sept. 1995), pp. 79–90. IFIP WG 10.4, preliminary proceedings.

Hayes, B. 1995. Waiting for 01-01-00. *American Scientist. 83*, 1.

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

Lehman, M. M. 1980. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE. 68*, 9, 1060–76.

Murphy, G. C. and Notkin, D. 1995. Lightweight source model extraction. In *SIGSOFT '95. Third ACM SIGSOFT Symposium on Foundations of Software Engineering* (Oct 1995).

Murphy, G. C., Notkin, D., and Lan, E. S.-C. 1996. An empiricial study of static call graph extractors. In *Proceedings of the 18th International Conference on Software Engineering* (March 1996), pp. 90–99.

Parnas, D. L. 1994. Software aging. In *16th International Conference on Software Engineering (Cat* (May 1994), pp. 279–87.

Scherlis, W. 1994. Boundary and path manipulations on abstract data types. In *IFIP Transactions A (Computer Science and Technology)* (Sept. 1994), pp. 615–20.

VanInwegen, M. 1996. The machine-assisted proof of programming language properties. Ph. D. thesis, University of Pennsylvania.

Wing, J. 1990. A specifier's introduction to formal methods. *Computer*, 8–22.