Issues in Performance Certification for High-Level Automotive Control Software

Bruce W. Weide[1] Paolo Bucci[1] Wayne D. Heym[1] Murali Sitaraman[2] Giorgio Rizzoni[3]

[1]*Computer Science and Engineering*
The Ohio State University
Columbus, OH USA
{weide.1,bucci.2}@osu.edu
w.heym@ieee.org

[2]*Computer Science*
Clemson University
Clemson, SC USA
murali@cs.clemson.edu

[3]*Center for Automotive Research*
Mechanical Engineering
The Ohio State University
Columbus, OH USA
rizzoni.1@osu.edu

Abstract

High-level supervisory control software for automotive applications (e.g., drive-by-wire) presents many challenges to making performance guarantees, which are a necessary part of the software's certification for deployment. The features of such systems demand that a compositional, or modular, approach to reasoning about performance be devised and applied. We discuss one such analytical approach as an alternative to simulation and testing.

1. Introduction

Modern automotive software systems are integrated assemblies of independently specified and developed subsystems, or components. The design process for such software assemblies necessarily relies on interface contract specifications that summarize functional and performance behavior of components without revealing internal implementation details. Why? Even if interface contracts were not a huge technical advantage for component-based systems—and they are [21]—they are a practical requirement: integrators of automotive software systems simply cannot rely on having source code for components provided by third-party suppliers.

At the same time, many features of automotive control systems are safety-critical [2,26]. There is a large body of work on software system safety in general, including studies of software problems that have led to human deaths (such as the infamous Therac-25 incidents [19]). The problems uncovered in such cases often turn out to be interface contract problems that have led to unsound reasoning about software behavior—either functional behavior or, in the case of real-time systems, performance behavior. The question then arises:

> What methods should be used to certify the correct behavior of an integrated assembly of software components, such as a high-level automotive control system, even when all the system's source code is not available?

Two of us (MS and BWW) attended this workshop last year to find out more about how this question was being addressed by those on the front lines of the automotive industry. We were struck by the fact that there seems to be one widely adopted approach to establishing any behavioral property of an automotive software system: simulation and testing. Unfortunately, there is no reason to believe that simulation and testing are ***sound*** ways to assess the behavior of complex software assemblies. Therefore, it is important to explore complementary analytical techniques that are made possible by building on considerable recent work on formalizing sound compositional reasoning about functional and, more recently, performance behavior [12,29,11,31,32,17,5,16] of software. This work can and should be applied in the context of automotive systems.

The purpose of this paper is to focus attention on one aspect of the automotive software certification problem, i.e., that of establishing real-time performance guarantees for high-level supervisory control software. We discuss why performance predictability—as opposed to simply "high performance"—is important for automotive applications, and relate this observation to the many technical issues involved in reasoning soundly about the performance of component-based software systems. Rather than presenting new technical results, we outline some of the main technical issues involved in compositional reasoning about performance, and provide

an overview of how the barriers to performance certification of component-based automotive software systems can be overcome. Given the page limitations of a workshop paper, we do this primarily by outlining the main ideas involved in the approach, and by citing recent supporting work that may be unfamiliar to automotive software engineers.

2. Background: Certification Techniques

The need for software that is both functionally correct and satisfies real-time performance guarantees is hardly unique to the automotive industry. One might hope, then, that experience from similar application domains would be immediately applicable. For example, avionics software shares with automotive software the need for certifiability. The U.S. Federal Aviation Administration (FAA) "establishes procedures for evaluating and approving aircraft software" in the U.S. [7], according to RTCA DO-178B "Software Considerations in Airborne Systems and Equipment Certification" [27]. However, RTCA DO-178B provides process-oriented guidance, not a prescriptive discipline for software design that supports either functional or performance verification. FAA certification of avionics software essentially requires a review of the software life cycle processes to verify that they conform to DO-178B, and a review of the software product and associated documents to verify that the approved set of processes was actually followed. Testing is a fundamental component (together with inspections, demonstrations, reviews, etc.) in the software process described by DO-178B. Thus, it is not surprising that testing plays a major role in avionics (as well as automotive) software validation.

It is well known, however, that testing—even with real code, let alone a simulation of it—cannot be used to establish either the functional or performance correctness of software systems. The purpose of testing is to reveal defects, not to show correctness or to enable prediction about situations not tested. As Dijkstra eloquently explained, "Program testing can be used to show the presence of bugs, but never to show their absence!" [4].

Notably, no comparable U.S. government agency certifies automotive software or software processes. The Motor Industry Software Reliability Association [22], a U.K.-based organization, offers process guidance similar to that of RTCA DO-178B for avionics software. Certification relies on testing; MISRA mentions an alternative approach, formal methods, as an "emerging" technology. The EU auto industry, including Audi, BMW, DaimlerChrysler, Porsche, and VW, is also cooperating on a new software quality effort [13,34]. No such wide-ranging program exists in North America.

In summary, software certification efforts based on simulation and testing are inherently too weak to establish a high degree of confidence that software will behave correctly when deployed. Certification procedures that depend on testing are currently in place not because testing is the only or best technical approach to certification, but because formal analytical models of software behavior that support sound reasoning have not yet been fully demonstrated. As the next two sections explain, the issues in reasoning soundly about the real-time performance of automotive software systems are challenging. In principle, however, they can be addressed by formal methods for modular reasoning about component-based software performance as well as functionality. However, these methods have not yet been applied in the automotive software domain.

3. Low-Level vs. High-Level Control

In automobiles, examples of ***low-level (servo-loop) control systems*** include variable valve timing, cruise control, and anti-lock brakes. There are hundreds of other examples in domains ranging from microwave ovens to MRI scanners to avionics. The practical direction of industry has been to replace traditional mechanical and electro-mechanical control devices by (potentially) "tunable" digital devices. As noted below, there have been many advances in this realm, both theoretical and practical.

In contrast, ***high-level (supervisory) control systems*** include embedded software that performs real-time processing of input data arising not only directly from sensors, but also from command messages sent from other control subsystems. Outputs serve not only as direct inputs to actuators, but as command messages to other control subsystems and to embedded databases. Examples under development at the Center for Automotive Research (CAR) at The Ohio State University include real-time energy management for a hybrid electric vehicle, and drive-by-wire (all-electronic) vehicle control. The latter is especially challenging. The driver is still in charge under drive-by-wire, but software controls steering response, acceleration, and braking, among many other features of vehicle performance. Such systems are also called X-by-wire because there are so many aspects of the vehicle that eventually can and should be placed under embedded software control—*if* it is possible to ensure that the software works correctly, including meeting all real-time performance requirements. Even in the absence of government-mandated certification requirements, automotive manufacturers and integrators have strong economic incentives (not to mention a moral imperative) to make sure that their high-level control software works correctly.

4. Analytical Models of Control Software

How can either kind of control software be certified analytically? At the block-diagram level, both low-level and high-level controllers can be viewed as conventional feedback control systems. Unfortunately, the analytical methods that apply to analysis and design of low-level control systems do not match the details in the blocks of

a high-level system. One feature that distinguishes high-level control systems is that the "system" being controlled—sometimes another software component, sometimes a physical subsystem controlled by another software component—often cannot be modeled with the usual tractable mathematics of engineering, e.g., differential and difference equations. There may be a finite number of control modes, yet in each mode the system typically must behave in ways that are manifestly not appropriately modeled as finite-state. Instead, each of a finite number of control modes will itself comprise a significant software system that will be constructed from software components in much the same way as other modern information systems are designed.

Prior research using formal approaches in the design of embedded real-time software systems has focused primarily on low-level control software. Here, it is often possible to predict bounds on task execution time: the deadline of an individual software task is determined by the time constants of the physical system it is intended to control, and multiple tasks must be scheduled so that each meets its deadline. The groundwork was laid thirty years ago with a definition of the problem as a matter of "multiprogramming in a hard-real-time environment" [20]. A number of periodic tasks, each with its own deadline (resulting from application-level timing requirements), must be scheduled on a single processor. Given this information—possibly with constraints on precedence, minimum interarrival times for sporadic tasks, and various complications added over the years—the problem is to find a static schedule that allows all tasks to meet their deadlines [3]. The current favorite approach, rate monotonic analysis (RMA) [28,15,18,23], is used in practice to address exactly the kind of problem facing automotive software engineers when scheduling *low-level* tasks on an electronic control unit (ECU).

5. Issues Specific to High-Level Control

Two features of high-level control software raise issues in directly applying such analytical models and their associated design techniques. First, like other algorithms for the problem, RMA assumes that each task in each cycle has an execution time that is bounded above by a known constant. For low-level servo-loop control, this is often a reasonable assumption that is readily shown to be true. For high-level control, it is not. In other words, the problem is not determining such a constant (e.g., as in [33,8]), but its very existence. That is, it is important to be able to predict *whether* a high-level control task can *always* be completed in bounded time, independent of the values of the variables involved in the computation. This requires sound predictive reasoning about software performance using an analytical model, and cannot be established by simulation and testing. Moreover, when the values of program variables may affect performance, a prerequisite to soundly predicting the performance of a computation is the ability to soundly predict the values that all variables will (or may, in the case of nondeterminism) have at any point in it. In other words, certification of functionality is part and parcel of certification of performance of high-level control software; an approach that tackles the performance problem alone is doomed.

Obviously, then, whenever possible it would be helpful to design software components to satisfy the bounded-execution time assumption. This reveals a second source of difficulty for models such as RMA, one that has received less attention than the first. How can software components that are useful in high-level control software be designed to satisfy the bounded-execution-time assumption? We know of no prior work directly on this problem, though various well-known algorithmic methods that amortize computation costs over multiple operations of a single component are promising. For example, a component interface contract design technique called "recasting" is sometimes able to achieve this [35,30]. However, for the rest of this paper, we focus on the more typical case for high-level control software: components where at least one operation takes time (or memory) that depends on the values of program variables in such a way that it is not bounded above by a constant.

The importance of being able to reason about tight bounds on performance is also somewhat different for high-level automotive control systems than for similar systems in manufactured goods with lower production volumes and higher per-unit costs (e.g., medical equipment and aircraft). A subtle reason for these differences is that auto manufacturers can save millions of dollars per year by cutting the costs of ECUs by a few dollars per vehicle, and hence are strongly tempted to do so. The economics of this competitive business make it inefficient to use a processor with significantly more speed or memory than necessary—let alone something as powerful as a standard PC. The developers of CT scanners and jet planes, for example, face different cost-benefit considerations. These differences mean that brute-force techniques that sometimes have been adopted to develop and to help certify the performance correctness of embedded software in such domains are not necessarily applicable to automotive control systems.

In addition, the system architecture is not something that an automotive software engineer can dictate. Automotive control systems are hierarchical, as a matter of design convention, industry standards, and ultimately common-sense engineering principles. For instance, in CAR's current experimental hybrid vehicle energy management system, a number of off-the-shelf subsystems being integrated are treated as "black boxes"; the combustion engine and electric motor are examples. Along with numerous sensors and actuators, these subsystems are connected, to each other and to the processor that runs the high-level control software, via an industry-standard network [1,24,25]. A current favorite is *CANbus*, or "controller area network bus", for which various high-level protocols are being standardized [6].

The combustion engine, for instance, comes from the manufacturer with a CAN-accessible interface contract through which it can be issued commands, e.g., to generate a certain torque. The electric motor subsystem is similar. There are a number of sequential tasks to be performed in each "cycle". These tasks can be statically scheduled in a simple fashion on one control processor.

Because of the "black box" nature of high-level control software components, any workable method for performance prediction in this domain must be able to deal with the abstraction that arises in component interface contracts. That is, often it is not possible to know low-level details of all components; one must have available, and be able to use, only summary information about component behavior in order to predict client system behavior. We return to this point in Section 6.

It is technically true that such a control system *as a whole* can be viewed as a sophisticated distributed computing system with multiple communicating processes running on multiple processors. However, many of the software problems facing the automotive control engineer are *not* the fundamental problems facing the distributed computing community in computer science. The issues are different: (1) to integrate a number of off-the-shelf components with their own interface contracts into a coherent sequential program, i.e., into a single task; (2) to determine the minimum processor speed needed to run a set of such tasks, one after another and typically in round-robin fashion, while permitting all the sequential tasks to meet their local deadlines; and (3) to determine the minimum memory the ECU must have in order to do this.

A typical industry development process to address these problems has been emulated by developers at CAR. It involves three phases. First, control software is prototyped on a special-purpose hardware/software platform and executed against a vehicle simulator. The prototype control software might be written in C, or in a graphical language with comparable linguistic features such as LabVIEW [14] or ASCET-SD [24]. Second, after testing and modifications, the more polished prototype control software is attached to a physical vehicle for "calibration", and then tested more. Finally, the prototype software is ported to a processor that will be used in production, and tested still more [9]. In the final phase in particular, it is not easy to match the performance correctness requirements of automotive control software to the speed and memory of hardware that might run it on a production vehicle.

In summary, high-level automotive control software poses many difficult and largely unexplored issues for certification of performance. Even with a technique other than simulation and testing to try to assure functional correctness, predictability of performance (as opposed to sheer high performance) of each component in isolation remains a serious problem. Compositionality of analysis in the face of component assembly, so component properties described and verified in the prediction phase are preserved when those components are assembled into a system, is a major problem. As noted earlier, it is well understood in the software engineering community that not only functional correctness, but also performance correctness, simply cannot be established by testing. All conditions that might be encountered by production vehicles cannot be anticipated during development and testing. This leads to an inevitable tension between building in an engineering cushion by oversizing the processor and memory "just in case", and the desire to keep costs down to make the product more competitive.

The next section explains how an approach we have developed for general-purpose component-based software—but have not yet applied to high-level automotive control software—suggests that these challenges to automotive software performance certification can be met with analytical methods.

6. Performance Certification

A component-based framework for certification of performance correctness of large software systems—especially those for which no single developer has access to all the source code—must permit a ***modular analysis*** [29,31,32]. This means that it should be possible to reason about the performance behavior of an assembly of components by using performance specifications of the components, without the need to re-analyze the implementation of each component in each use. We are able to give only an outline of our approach to doing this, but provide several references to previous papers that explain many of the details.

First, a limitation: the approach does not handle one special aspect of performance. An ideal framework for doing modular analysis of performance should, in principle, ultimately allow one to account for hardware optimizations such as caching, pipelining, and memory management support. There has been some interesting work in this area, such as [33,8], which is largely orthogonal to ours. It gives probabilistic bounds, though, and does not deal with abstraction in component interface contracts and is therefore not modular. As noted below, the problem of performance prediction for high-level supervisory control software is difficult even in an idealized computational model that factors out such effects. We hope that hardware optimizations can be factored in later through looser bounds on the precision of performance specifications, rather than being considered impediments to modularity. If such bounds turn out to be terribly pessimistic and hard-real-time guarantees are not needed in some cases, then results involving worst-case execution time in the presence of caches, pipelining, etc., might be integratable with our analysis approach.

Two kinds of problems must be addressed: (1) how to specify formally both execution time and memory usage; and (2) how to use such specifications to reason modularly about both execution time and memory usage of assemblies of components, based on component

performance specifications of like kind. Semantics and proof rules for execution time are intuitively straightforward in an idealized model. But significant technical difficulties have, until recently, impeded specification of both execution time and memory usage, as well as verification of the latter.

Reasoning about the functional behavior of component-based systems relies on the existence of an abstract mathematical model for each data type in an interface contract specification. Using this mathematical model, it is possible to summarize and thereby explain the functionality of any correct implementation of the contract. The performance specification for each component implementation must be expressed in similar kinds of conceptual terms that are understandable to client programmers without a complete knowledge of its internal details.

The performance specification problem naturally becomes more complex in the face of non-determinism and dynamic storage allocation. Both should be allowed as routine in high-level control software because they can make functional correctness easier to establish than if artificial design constraints require software developers to avoid them. Yet, performance specification is difficult even for simple but realistic components that do not involve such features. One problem we have addressed is that the execution time and memory usage behavior of a component really must be expressed in terms of the *abstract values* of the variables/objects involved, not merely their *sizes* as in algorithm analysis textbooks [32]. For complex user-defined types arising in high-level supervisory control software, this distinction must be made or the resulting performance estimates are useless.

The distinct nature of execution time (which is used up) and memory (which is recycled) is also an issue because it means that the calculi for verifying these two aspects must be different. It is clear how to account for time in an idealized computational model without, e.g., caching or other hardware optimizations: the execution time of two statements executed sequentially is the sum of the execution times of the individual statements. It is not obvious how to account for memory usage, which is manifestly not additive.

Intuitively, the idea for how we address the compositionality problem for performance certification is to notice that performance of a component *C* is a function of the performance of the components on which it is built. Given "performance profiles" for these lower-level components [16] and "performance semantics" for the statements of the programming language, it is possible to calculate a performance profile for *C as a function of the performance profiles of its subsystem components*. Modular reasoning depends on the fact that this may be done once from an analysis of *C*'s code, i.e., not once for each possible choice of subsystem component implementations. The performance specifications of the not-yet-determined subsystem implementations are then assumed (by induction) to be known to bound their respective performance profiles. This approach leads to proof rules for verifying the performance profile of *C* against its performance specification (justifying the induction hypothesis by moving the completed verification "up one level" in the system). The remaining issue is to establish appropriate performance specifications for the lowest-level components that are implemented using only programming language built-ins (the base cases in the induction).

In order to address performance specification expressiveness, we have developed the use of "intermediate models" to write tight performance bounds. In general, abstract models used in behavioral specifications need to be augmented in different ways, depending on the precision required in performance specifications [29,36]. The idea is to introduce a new, third kind of program unit, different from both an interface contract specification and a component implementation. This module describes an augmented model for a type that is used to specify execution time and memory usage.

The basic idea we have pursued to address the memory usage verification challenge is to separate "standing" and "transitional" memory usage. The reasoning system must be able to show that, before an operation is called, the (standing) memory used by the representations of objects in scope, together with a specified (transitional) memory requirement needed only during the operation's execution but no longer after it terminates, lies within the total available memory capacity of the system [17].

7. Conclusion

High-level automotive control software presents interesting challenges for software engineering researchers, perhaps even more than embedded systems in general because of some special features of the automotive domain. The immediate need to get systems built, however, should not deter the automotive industry from exploring software certification approaches other than simulation and testing—new approaches that promise to deliver sound reasoning about both functional *and performance* behavior of high-level supervisory control software.

We thank the referees for several useful suggestions. We also gratefully acknowledge financial support from the U.S. National Science Foundation under grant CCR-0113181.

8. References

1. Baumann, B., Rizzoni, G., and Washington, G. Mechatronic design and control of hybrid-electric vehicles. *IEEE/ASME Transactions on Mechatronics*, March 2000.
2. Broy, M. Automotive software engineering (panel summary). In *Proc. 25th Intl. Conf. on Software Engineering*, IEEE, 2003, 719-720.

3. Byrnes, D.D., and Weide, B.W. Static scheduling of hard real-time control software using a complex periodic execution model. In *Proc. IEEE Workshop on Parallel and Distributed Real-Time Systems*, IEEE, 1993.
4. Dijkstra, E.W. *Notes on Structured Programming*. Technical University Eindhoven, T.H.-Report 70-WSK-03, 1970.
5. Dunn, W.R. Designing safety-critical computer systems. *Computer 36*, 11 (Nov 2003), 40-46.
6. Etschberger, K. *CAN — Controller Area Network, 2nd edition*. Hanser-Verlag, Munich, 2000.
7. Federal Aviation Administration. *FAA Order 8110-49: Software Approval Guidelines*, 2003, http://av-info.faa.gov/dst/reference.htm.
8. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., and Wilhelm, R. Reliable and precise WCET determination for a real-life processor. In *Proc. First International Workshop on Embedded Software (EMSOFT 2001) — LNCS 2211*, Springer-Verlag, 2001, 469-485.
9. Gilstrap, M., et al. Design and development of the 2003 Ohio State University FutureTruck. In *Proc. SAE Intl. Congress and Exposition*, SAE, March 2004.
10. Hammond, K., and Michaelson, G. Hume: a domain-specific language for real-time embedded systems. In *Proc. GPCE '03—Generative Programming and Component Engineering*, Springer-Verlag, 2003.
11. Hehner, E. C. R. Formalization of time and space. *Formal Aspects of Computing*, Springer-Verlag, 1999, 6-18.
12. Hooman, J. *Specification and Compositional Verification of Real-Time Systems*. Springer-Verlag LNCS 558, New York, 1991.
13. Jersak, M., et al. Formal methods for integration of automotive software. In *IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE'03 Designers' Forum),* IEEE, 2003.
14. Johnson, G.W., Jennings, R., and Jennings, R. *LabVIEW Graphical Programming, 3rd edition*. Mc-Graw-Hill, Upper Saddle River, NJ, 2001.
15. Klein, M.H., *et al. A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer, Boston, 1993.
16. Krone, J., Ogden, W. F., and Sitaraman, M. *Profiles: A Compositional Mechanism for Performance Specification*. Technical Report RSRG-04-03, Department of Computer Science, Clemson University, Clemson, SC, June 2004, 24 pages; available at http://www.cs.clemson.edu/~resolve.
17. Krone, J., Ogden, W. F., and Sitaraman, M. Modular verification of performance constraints. *Proc. ACM OOPSLA Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, 2001, 60-67.
18. Lehoczky, J.P. Real-time resource management techniques. In *Encyclopedia of Software Engineering*, J. Wiley & Sons, New York, 1994, 1011-1020.
19. Leveson, N., and Turner, C.S. An investigation of the Therac-25 accidents. *Computer 26*, (July 1993), 18-41.
20. Liu, C.L., and Layland, J.W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM 20*, 1 (1973), 46-61.
21. Meyer, B. Applying design by contract. *Computer 25*, 10 (Oct. 1992) 40-51.
22. Motor Industry Software Reliability Association. *Development Guidelines for Vehicle Based Software*, 1994, http://www.misra.org.uk; also published as ISO/TR 15497.
23. Naghibzadeh, M. A modified version of rate-monotonic scheduling algorithm and its efficiency assessment. In *Proc. 7th Intl. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, 2002, 289-294.
24. Paganelli, G., Ercole, G., Brahma, A., Guezennec, Y. G., Rizzoni, G. General supervisory control policy for the energy optimization of charge-sustaining hybrid electric vehicles. *JSAE Review 22*, 4 (2001), 511-518.
25. Paganelli G., Guezennec, Y. and Rizzoni, G. Optimizing control strategy for hybrid fuel cell vehicles. SAE Paper 2002-01-0102, 2*002 SAE International Congress*, Detroit, MI, March 2002.
26. Pisu, P., Silani, E., and Rizzoni, G. Supervisory robust control of hybrid electric vehicles. In *Proc. ASME IMECE 2003*, ASME, 2003.
27. Radio Technical Commission for Aeronautics. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, 1992, http://www.rtca.org.
28. Sha, L., Klein, M.H., and Goodenough, J. Rate monotonic analysis for real-time systems. In *Foundations of Real-Time Computing: Scheduling and Resource Management*, Kluwer Academic Publishers, Boston, 1991, 129-155.
29. Sitaraman, M. On tight performance specification of object-oriented software components. In *Proc. of the 1994 Intl. Conf. on Software Reuse,* IEEE, 1994, 149-157.
30. Sitaraman, M., Weide, B.W., Long. T.J., and Ogden, W.F. A data abstraction alternative to data structure/-algorithm modularization. In Jazayeri, M., Loos, R.G.K., and Musser, D.R., eds., *Generic Programming*, Springer-Verlag LNCS 1766, 2000, 102-113.
31. Sitaraman, M., Kulczycki, G., Krone, J., Ogden, W.F., and Reddy, A.L.N. Performance specification of software components. In *Proc. SSR '01: 2001 Symp. on Software Reusability (ACM SIGSOFT Software Engineering Notes 26, 3)*, ACM, 2001, 3-10.
32. Sitaraman, M. Compositional performance reasoning. In *Proc. 4th ICSE Workshop on Component-Based Software Engineering*, IEEE, 2001, 98-101.
33. Theiling, H., Ferdinand, C., and Wilhelm, R. Fast and precise WCET prediction by separate cache and path analyses. *Real-Time Systems 18*, 2/3, 2000, 157-179.
34. Tindell, K., Kopetz, H, Wolf, F., and Ernst, R. Safe automotive software development. In *IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE'03 Designers' Forum),* IEEE 2003.
35. Weide, B.W., Ogden, W.F., and Sitaraman, M. Recasting algorithms to encourage reuse. *IEEE Software 11*, 5, 1994, 80-88.
36. Weide, B.W., Ogden, W.F., and Sitaraman, M. Expressiveness issues in compositional performance reasoning. In *Proc. 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, 2003, 85-90.