Formal Methods in Embedded Design

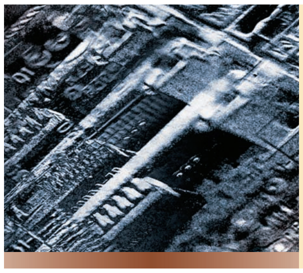
Steven D. Johnson, Indiana University

For more than two decades, applied formal methods have remained the unexplored frontier of embedded system design—just beyond the reach of practice. There have been inroads, certainly, but no sign of a revolution—even a quiet one—in industry. Considering the rapid progress of technology over this time, the dynamic expansion of applications, and the meager investment in formal methods, just keeping pace with the advancing frontier speaks pretty well for the research.

Having watched these trends over the years, I would not forecast any dramatic changes in practice. We should be looking for a sea change, not a revolution. Formal methods have a cumulative impact, reflected in languages and "informal" methods as much as in automated reasoning tools.

Nevertheless, I see their use in embedded systems design accelerating in the coming five years, especially if we consider the full spectrum of approaches described in the "Formal Methods Terminology" sidebar. System developers must watch this trend closely and foster appropriate long-term strategies.

The most important strategic investment is in expertise. The big winners will be the successful risk takers, but the risks are still significant enough to make expertise more important than technology.



The future benefits of using formal methods in embedded system design warrant investment in the underlying expertise.

A BEACHHEAD IN PRACTICE: HARDWARE VERIFICATION

In the early 1980s, a branch of formal methods research coalesced around digital hardware. Algorithmic breakthroughs and groundbreaking case studies, as well as promotion by CAD developers, piqued industry's interest. Improved automation, along with some well-publicized design failures—most notably, perhaps, an error in the Pentium I's arithmetic unit—helped motivate a few large companies to establish in-house formal verification groups in the 1990s. Microprocessor manufacturing led the movement, with telecommunications and multiprocessing also showing substantial advances.

Success has come at all automation levels—from broad-based use of equivalence checking to pervasive applications of model checking to targeted applications of theorem-proving in areas such as floating-point arithmetic.

Hardware-like systems first

Embedded systems that share qualities with hardware systems are the first candidates for use of formal methods. Deeply embedded control systems are one example. They have a dedicated real-time function and are difficult to physically modify. Smart cards, on the other hand, present extreme security and privacy requirements.

Because formal methods are based on mathematical models, physical similarities with hardware are less important than abstract similarities. Being "hardware-ike" has more to do with design concepts than with form or function. Specifically, hardware design practice employs a synchronous methodology with relatively low levels of data and control abstraction. In combination, these features are compatible with direct, compositional formalization, paving the way for effective decision procedures and useful proof strategies.

The early adopters of formal methods for embedded system design will be those developers whose design *practice* is already hardware-like—hard real-time control is an example. Applications that can adopt more stringent, uniform design methods will be able to benefit from formal methods by doing so.

Programming language dilemma

The notion of making embedded software design more hardware-like raises a problem. Industry is under pressure to use prevailing programming paradigms to populate its workforce. Unfortunately, these paradigms and languages, inspired by geographically distributed systems, are ill-suited for both embedded applications and effective formal analysis. Java and C++ are asynchronous object- and event-oriented languages. Embedded system design calls for synchronous process-oriented languages.

The prominent hardware simulation languages, VHDL and Verilog, are also asynchronous; so are some system design languages, such as SystemC and

SpecC. To enable automated reasoning, the hardware simulation languages superimpose synchronous dialects for verification and synthesis. Developers must apply the same methodology in embedded software design. Unfortunately, this will delay the deployment and diminish the effectiveness of formal methods.

A FOOTHOLD IN CRITICAL SYSTEMS SOFTWARE

In the 1970s, proponents found potential applications for formal methods in embedded software for avionics, space exploration, nuclear control, and so on, where the consequences of a design error are extreme. Simulation and testing cannot demonstrate the reliability these systems require. Rigorous mathematical analysis is essential. Validating this analysis and correlating it to actual implementations suggested uses for automated reasoning. Of course, this rationale begged the question of whether anyone knew how to perform the analysis in the first place.

Justifiable indignation of practitioners about the naivete of formalists, supercilious philosophical debates, and internal bickering among researchers all generated bad publicity, but formal methods have nonetheless made steady progress in critical systems design. Today, modeling foundations are converging, unified tools and methods are emerging, and products are in development whose design and ceritification entail the use of formal methods.

Regulatory agencies and government-sponsored research, particularly at NASA, have provided the initiative for these efforts. For example, government advocates pushed for formal verification as an auxiliary means of verifying and validating avionics systems seeking the highest levels of safety certification.

Despite progress, none of this work will lead to mandated use of formal methods any time soon. For that to occur, effectiveness, practicality, and a standard of practice must emerge. Like the situation with hardware in the 1980s, industry has so far been reluctant to get involved.

Formal Methods Terminology

Formal methods are often described as the application of mathematical rigor to system design and implementation. The description is fair in some respects, but off the mark. In this context, "formal" refers to the direct or indirect use of a *logical formalism* as the basis for computer-assisted reasoning. Such formalisms include a syntactical grammar, axioms, and symbolic manipulation rules. There is at best a marginal relationship between the notion of mathematical rigor and the mechanical rigidity of automated reasoning.

Hardware and software engineering sometimes draw different distinctions between *verification* and *testing*. In the hardware literature, the term *formal verification* distinguishes proof-based, exhaustive verification from the simulation-based, conventional verification techniques employed in practice. In software, verification often connotes formal analysis.

Verification is one of many modes of reasoning about design correctness. *Formal synthesis* is an example of an alternative mode of reasoning by which correct implementations are algebraically derived from their specifications. Constructions, constraints, refutation, and so on add further texture to the taxonomy. I apply the term "formal methods" to the entire genre of automated, interactive reasoning applied to system design.

LET THE RACE BEGIN

For formal methods to contribute significantly to safety-critical systems, at least one major industry must take the plunge. John Rushby has argued cogently for the automotive industry as a likely candidate ("Bus Architectures for Safety-Critical Embedded Systems," *Proc. 1st Workshop on Embedded Software* [EMSOFT 2001], Springer-Verlag, 2001, pp. 306-323).

Automotive embedded control architecture is undergoing a fundamental transition, making it more vulnerable to design errors. Until recently, dedicated controllers communicating over private channels implemented critical control functions. Because of their isolation from one another, these subsystems were reasonably well insulated from systemwide failures.

Automotive control complextity has reached the threshold at which ad hoc system architecture must give way to more cost-effective integrated solutions. Integrated systems depend on shared resources—specifically, a common communication network. Consequently, every process becomes vulnerable to failure in any component, making global fault containment a necessity.

The aerospace community has grappled with this problem for many years, so a foundation exists to work from. The cost of achieving reliability for airborne systems is prohibitive for automobiles, so a major market exists for derivative products.

All this should stimulate both academic and entrepreneurial activity. In a competitive industry, some company is likely to adopt a strategy of investing in formal methods, if only to hedge against future rivals. So the race begins.

EXTENDING THE TERRITORY

In his keynote presentation at the 2001 European Joint Conferences on Theory and Practice of Software, Michael Lowry surveyed NASA's considerable experience with embedded systems for space missions (www-etaps.imag.fr/Invited/invited.php). He pointed out that failures are typically consequences of unanticipated scenarios and Byzantine subsystem interactions. Lowry's concept of "interaction" is a broad one, including not only component interactions within a system or its environment but also interactions between engineering tasks or management levels. He observes that as systems become larger and less regular, design paradigms must strike a different balance between analytic and constructive approaches.

We cannot limit automated reasoning to analytic approaches of verification and property checking. It also includes iterated design, refinement, and synthesis, with comparable and compatible automation. Unfortunately, tool support for constructive formal methods lags far behind the support for model checking and theorem proving, and this gap is another impediment to advancement in practice.

Although more limited in scale, the experience in software suggests new pathways for incorporating formal methods. The implementation of these pathways is a matter of speculation, but I offer some guesses.

Requirements analysis

Industrial pilot studies have repeatedly demonstrated the value of formalizing early specification stages. Frail tools, insufficient expertise, or problems integrating with existing design flows inhibit subsequent adoption of formal specification paradigms. In spite of these problems, formal requirements specification and analysis is the obvious way to initiate formal methods use.

Reliable infrastructure

The coming generation of certifiably fault-tolerant networks, hardware, and middleware for automotive and avionics applications already entails formal methods use. In these and other safety-related components, formalization translates directly to commercial value in terms of certifiability.

Conformance validation

A reliable infrastructure permits system design and testing at a higher abstraction level. Developers can apply formal methods to a symbolic mathematical representation and use it to reason formally about applications or their programming interfaces.

Formal software verification involves automated model extraction to eliminate extraneous detail from target implementations and thus enable model checking. Alternatively, a synthesis tool starts with a protocol model and adds implementation details. Tools for these tasks are the subject of current research, with some products now becoming available.

Formal intellectual property

As the use of formal models increases, so does pressure on vendors to include them with components. Providing formal specifications suitable for automated reasoning adds value to products. It can also indemnify producers against component misuse. Producers can avoid blame for system failures if they can formally verify correct functionality.

EDUCATION IS THE LIMITING FACTOR

At the beginning of this essay I said that the embedded systems industry needs to be watchful but is not facing a widespread increase in the use of formal methods. There are some technological impediments, but the root issue is a severe shortage of expertise.

I have outlined ways that I believe formal methods may infiltrate mainstream practice. Missing from the list is the massive adoption of highly automated verification tools that the hardware industry experienced. Designers will use existing hardware tools wherever they can profitably do so, of course, but there is no evidence to suggest that something as transparent as equivalence checking will be useful in embedded system design. The hard problems are global in nature.

Designers can rarely use formal methods out of the box. They must adapt even the most automated tools to the design context. This requires not merely tool skills, but also a conceptual grasp of logical modeling, mathematical manipulation, abstraction, decomposition, concurrency, and so on. Sadly, the computer science and engineering curriculum provides too little exposure to these concepts, and what exposure it does offer follows paradigms that are poorly suited for embedded applications.

We need to close the loop between practice and education, but we don't have enough interested practioners or knowledgeable educators to prime the pump. The situation foretells a gradual change in practice, starting with the most exacting applications and dependent on continued government sponsorship.

Perhaps a consortium of industrial interest in formal methods will develop, as happened with VLSI in the 1980s. Even were this to happen, however, it would take the educational establishment a number of years to respond with significant numbers of graduates. Formal methods constitute a difficult major, demanding not only an engineering background but strong aptitude in mathematics and tool proficiency. The alternative of retraining practicing designers is daunting without a proper conceptual foundation.

For some parts of the industry, the time to move forward into formal methods is now, but most can afford to wait and see what develops. It is only time to start looking at how formal methods will fit their needs. ■

Steven D. Johnson is a professor of computer science in the Indiana University College of Arts and Sciences. Contact him at sjohnson@cs.indiana.edu.

Editor: Wayne Wolf, Dept. of Electrical Engineering, Princeton University, Princeton, NJ, 08544-5263; wolf@princeton.edu