# Integrating Formal Techniques for the Development of Embedded Software (White Paper)

Insup Lee, Rajeev Alur, Carl Gunter, Sampath Kannan, Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania

Elsa Gunter
Computer Science Department
NJIT

POC: lee@cis.upenn.edu

November 2, 2001

**Abstract**

Formal methods have been available for about three decades. Although much progress in developing tools based on them have been made, their use has been limited by the expense (i.e., time, computational resources, and expertise) required to employ them. However, it is not possible to envision the development of high-quality embedded software without using such techniques in the future. In this white paper, we explain the problems with the current approach and described an integrated framework of that allows multiple uses of the same requirements and specification artifacts to increase the payoff of adapting formal techniques. We identify several research areas to facilitate the integrated use of formal methods.

## 1 Introduction

An embedded system consists of a collection of components that interact with each other and with their environment through sensors and actuators. Embedded software is used to control these sensors and actuators and to provide application-dependent functionality. Examples of embedded systems can be found in manufacturing, automotive controllers, avionic systems, PDAs, robots, and medical devices.

Embedded systems have been developed traditionally in an ad-hoc manner by practicing engineers and programmers. Knowledge of system behavior and functionality is contained in the mind of the domain-expert designer, and is only imperfectly captured and translated into embedded system products by the engineer and programmer. The existing technology for embedded systems' construction does not effectively support the development of reliable, robust, and reusable embedded software. When embedded systems are used in safety-critical applications, it is crucial to guarantee the absence of logical and timing errors.

There are two major factors that make embedded systems harder to develop. First, the software complexity of embedded systems has been increasing steadily as microprocessors become more powerful. To mitigate the development cost of software, embedded systems are being designed to flexibly adapt to different environments. The requirements for increased functionality and adaptability

make the development of embedded software more complex and error-prone. Second, embedded systems are increasingly networked to improve functionality, reliability, and maintainability. Networking makes embedded software even more difficult to develop, since composition, abstraction, and testing principles are poorly understood.

A variety of techniques are used to ensure the extra degree of quality (i.e., robustness and reliability) expected from high-assurance embedded software. For example, processes for software development can be carried out under more stringent rules, more code inspections can be ordered, test suites can be more numerous, simulations of the software's environment can be used to provide more complete testing, and field beta testing can be more extensive.

Although simulation and testing are the most widely used techniques, they are known to be incomplete and very expensive for systems that require high-quality assurance. A complementary and increasingly promising alternative to simulation and testing is the approach of formal modeling and analysis. Formal methods are a collection of techniques that allows the abstract model of software to be subjected to forms of mathematical analysis. Formal methods have been available for about three decades; their use has been limited by the expense (time, computational resources, and expertise) required to employ them, but they have been used in a significant number of applications involving high-security and safety-critical systems.

## 2 The Integrated Approach of Formal Techniques

Since it is believed that no one formal method will satisfied all the needs, it is necessary to develop an integrated-use methodology for formal techniques that can facilitate the development of reliable embedded systems. Such a methodology can be followed to complement the traditional software engineering processes. In particular, we should aim to exploit an 'end-to-end' support for the software engineering process. The main idea is that many formal methods are intended for use at one stage in the usual software engineering process (requirements capture, development, testing, and deployment), but their benefit can only be appreciated when understood across multiple phases. For instance, formal specification techniques may seem burdensome in requirements capture, but this burden is often more than offset by improved reliability and documentation, rapid prototyping and deployment. Note that most formal methods in current use deal with the early phases of the software engineering process and do not analyze deployed implementations. It is our position that there should be more work on the techniques such as test generation from specifications, run-time monitoring and checking based on specifications, to effectively deal with this end of the process.

**The Envisioned Approach to Integration.** Figure 1 shows the overall structure of the HASTEN (High Assurance Systems Tools and Environment) environment. A software engineering process is centered around the development of two entities, requirements artifacts and system artifacts (shown in the middle column of Figure 1) and the validation of system artifacts with respect to requirements artifacts. Requirement artifacts, initially constructed informally through the requirement elicitation, are gradually refined into more rigorous representations. System artifacts can range from design diagrams such as UML diagrams, to prototypes and specifications, to executable code. Each of them is developed to satisfy some aspects of the requirements. As shown in Figure 1, techniques such as prototyping, verification, testing, and monitoring can be used to evaluate that a system artifact meets its requirements during development and deployment of the system. Evaluation results are used as a feedback to modify the system artifacts, and sometimes the requirements. Any changes to the system and requirement artifacts, in turn, necessitates a new round of evaluation.

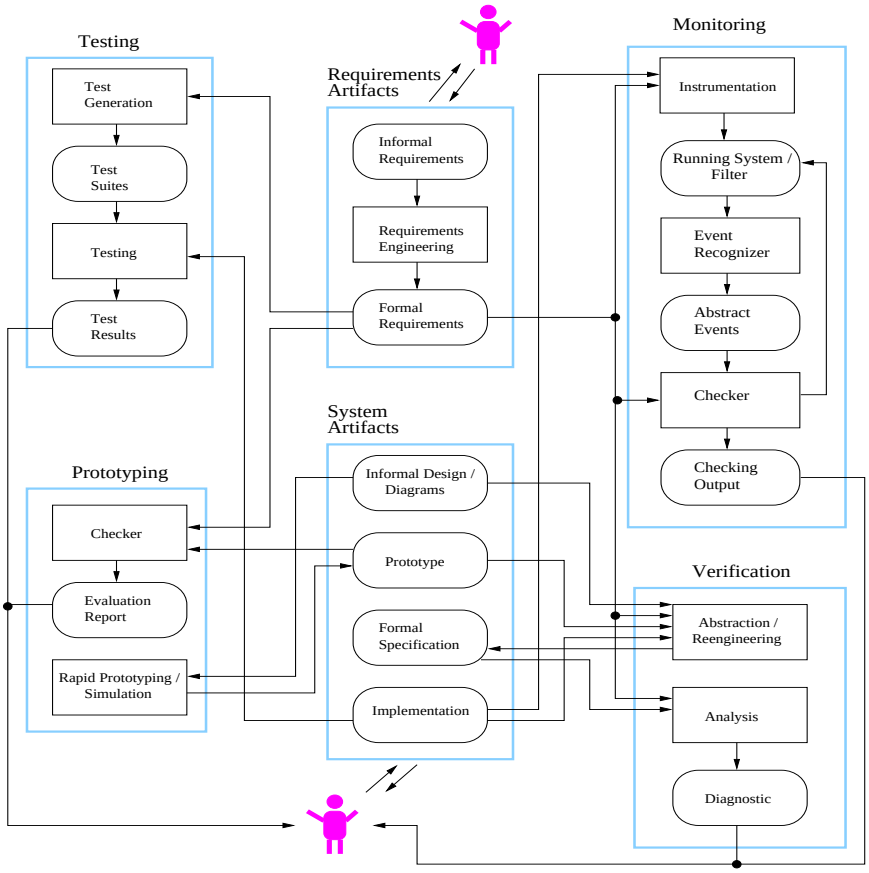


Figure 1: The HASTEN Environment

Although formal test generation, prototyping, verification, and monitoring can be very effective when applied independently, they have not been applied in unified fashion. We believe that it is important to develop a general framework where these different techniques can be more uniformly applied. With a uniform framework, one model of the system is specified and then can be used for both static and dynamic analyses. This would save the effort of developing multiple models since each technique requires its own model of the system. Furthermore, this would eliminate consistency checks between models used by different techniques since there is only one model. Each technique can be used to analyze different abstractions of the same model.

Another aspect is the integration of techniques 'end-to-end' across the software life cycle. The life cycle includes development, operation and maintenance activities and consists of a series of phases. The basic phases are user requirements specification, design of software requirements, architectural design, detailed design and coding, transfer of the software to operations, operations and maintenance. We believe that formal methods are most effective when they can span several of these basic phases. Figure 1 shows how we can apply analysis techniques to determine the correctness of a formal specification with respect to the requirements. Note that it may be necessary to apply several different formal techniques since different formal methods have complementary strengths. It is assumed that the formal specification is derived from informal designs or prototypes. Since complete analysis of a design may not always be possible, verification can be complemented by testing: test suites may be automatically generated from the requirements and can be used to validate both a design as well as an implementation. Finally, to ensure correctness

of an implementation during the operation phase, run-time checkers may be derived so that they can be executed by a run-time monitoring system.

## 3 Future Research Directions

We list several areas that need more research to support the integrated use of formal techniques.

**Capturing User Requirements and Environmental Assumptions.** The requirements phases are to specify how the system (consisting of hardware and software) and the user or environment are expected to behave and their interactions. The user requirements phase is the "problem definition phase" of a software product. Here, the scope of the product must be defined and the user requirements must be captured. At the same time, the operational assumptions on the environment should be clearly identified.

The software requirements phase is the analysis of a software product. This phase includes the construction of a model describing what the software has to do, rather than how to do it. The architectural platform should also be specified during this phase. The purpose of this phase is to identify and document the exact requirements of the embedded systems and its operating assumptions. It may be necessary during this phase to build a prototype to clarify the software requirements.

Research is needed to show how to elicit such requirements and assumptions, how to capture and specify them, how to analyze them, etc. It is important for the elicitation process to be intuitive for the domain experts, that the resulting requirements specification is amenable to refinement during subsequent stages of the design process, and also that it is amenable to analysis. Balancing these three aspects is one challenging problem since the goals are often contradictory. For example, formalisms that make the analysis easier are usually far removed from the problem domain. Furthermore, requirements start as informal description but need to be formalized to support analysis. Another challenging problem is how to facilitate the translation from informal description to formal specifications and to capture assumptions made by domain experts in informal requirements.

**Resource Constraint Expression.** For embedded system development, it is necessary to take into account real-time issues and resource constraints explicitly at the modeling phase, instead of dealing with them in an ad hoc manner after implementation. Many of real-time specification formalisms, such as timed automata, timed Petri net, and real-time process algebra, are mainly used for specification and analysis, and there is a big gap between specification and implementation. The gap needs to be closed so that alternative models, which are not feasible to implement, can be eliminated before the implementation phase. Furthermore, timing and resource constraints are needed to do schedulability analysis. The code generation phase requires scheduling to satisfy timing constraints and resource allocation to address resource constraints of embedded systems. Every step of the model refinement process from a high-level model down to executable code should take timing constraints into account. The goal should be to reduce the number of iterations of design cycles caused by the timing constraints.

**Probabilistic specification for real-time systems.** Probabilistic specification and analysis has many uses in the software development process. Most importantly, probabilities play an important role during requirements specification and early stages of design. The following factors yield probabilities in specifications: (1) during requirements engineering, probabilities are used to capture and formalize fault-tolerance and real-time requirements; (2) during design, the use of probabilities allows designers to represent underspecified components; (3) during modeling and

analysis of a system, its environment often has to be specified probabilistically, to account for lossy communication media, noise and interference between inputs, etc.

In addition, the notions of probabilistic or statistical testing are used by some development methodologies such as cleanroom. Support for probabilistic testing, e.g., through probabilistic test generation, can also be provided by incorporating probabilistic data into specifications.

**Abstraction Techniques.** One of the goals of the proposed framework is to be able to check consistency between behaviors of specifications from different development phases. In order to do this one must be able to map basic entities, or events, referred to in one specification to those addressed in the specification corresponding to a different phase. This is a difficult task because events in specifications at different levels during the software development, correspond to actions of varying complexity in the actual system; an atomic event at a higher level may correspond to a complex combination of events in specification that is closer to the implementation. Therefore, determining the compatibility of two different specifications depends crucially on the correctness of this "abstraction" mapping.

**Reconstruction of specifications using abstractions.** Another important use of abstraction is to reconstruct formal models from pseudo-code or, for smaller programs, directly from code. Given the current state of the art in formal specification, it appears impossible to construct a meaningful specification from pseudo-code automatically. In most cases, the user has to put in a certain amount of ingenuity to come up with the right abstractions; otherwise, the amount of detail contained in the code will make the specification impossible to analyze. Since the user interaction is necessary during the reconstruction process, we need to figure out how present information about the code to the user in such a way that would enable him to make the right decisions easier.

**Code generation from specifications.** Coding errors are a significant source of bugs in embedded systems. Automatic code generation is a way to eliminate this source of errors and ensure that the implementation conforms with the specification. EFSM (Extended Finite State Machines) specifications are particularly amenable to automatic code generation due to their simple structure. At the same time, code generation from high-level specifications is hampered by the fact that the same construct in the specification may have to be implemented differently in the code. This is especially true when it comes to communication mechanisms in distribute systems. Depending on the allocation of the system components to processors, the same communication mechanism may be implemented via shared variables if both components are on the same processor, or via point-to-point message passing if different processors are used. Thus, to make code generation efficient, the user has to supplement the high-level specification with additional information (similar to deployment diagrams in UML), and use this information in the code generation process.

Moreover, the nature of embedded systems presents an additional complication for automatic code generation. A wide variety of microprocessors is used to implement such systems, and their capabilities span a wide range. The same specification may have to be implemented in different ways to achieve maximum performance when run on different processors.

**Test generation from specifications.** Testing is the most common approach used to validate implementation. Recently there has been some work of generating tests from formal specifications such as EFSM. This is a promising approach since this would allow requirement artifacts created at the initial development phase be used to validate the implementation. However, much work is need to make this approach to be adapted in practice. They include the development of practical test coverage criteria, tools for test generation and optimization, test execution environment, as well as extending the approach to deal with hybrid systems.