

Supporting Model-Based Validation at Run-Time *

Insup Lee and Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
lee@cis.upenn.edu, sokolsky@cis.upenn.edu

May 19, 2003

1 Introduction

Dependability in embedded software systems can be significantly improved through the use of formal methods for the specification and analysis of systems. Formal methods treat system components as mathematical objects and provide mathematical models to describe and predict the behaviors and properties of these objects.

There are two essential components to formal methods used in software development. First, a formal method includes a formal language for describing a software artifact such as specifications, designs and source code. Second, a formal method supports mathematical reasoning about formulae in the language. Formal methods can be used during the requirements, architectural design, model specification, coding as well as testing and maintenance phases of software. Recent advances in formal methods may change future practice in software systems engineering. In particular, formal methods will be required in the areas of specification, design, and validation for the development and certification of safety-critical embedded systems.

Once a formal description has been produced, formal methods provide a variety of simulation and verification techniques for analyzing and reasoning about the specification. Simulation and verification require formalisms with well-defined semantics to ensure that there is no incorrect conclusion from misunderstood behaviors.

Verification is the process of showing that the possible behaviors of a system satisfy certain properties. A specification can be verified for completeness or consistency. It can also be verified to satisfy required safety and liveness properties using theorem proving or model checking techniques. Theorem proving in general requires much interaction by the human user, whereas model checking is more automated. In particular, a model-checker determines automatically whether a system satisfies a property given as a formula in temporal logic. This is done by exploring all possible states that the system can reach and then showing that the property holds in every state. Another advantage of model checking is that when a given property does not hold, it can produce a counter-example or witness that illustrates why the property fails to hold. This can be exploited for model-based test generation [6].

Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. There are three approaches:

*This research was supported in part by NSF CCR-9988409, NSF CCR-0086147, NSF CCR-0209024, ARO DAAD19-01-1-0473, and DARPA MOBIES F33615-00-C-1707.

testing, model extraction and verification, run-time monitoring and checking. Testing and run-time checking are top-down, whereas model extraction is bottom-up.

The purpose of model-based validation is to ensure that the implementation is consistent with the specification. Although formal verification analyzes all possible executions of a system, the analysis is performed on the specification of the system, not its implementation. Thus, there is no guarantee on the correctness of the implementation. In addition, the correctness proof itself can be incorrect. Furthermore, the existing verification techniques do not scale up well to handle large systems, and thus, it is not possible to verify implementations directly.

There are two approaches to use specification models to validate implementations. One approach is to use a specification model to generate tests and then apply them to an implementation. Another approach is to monitor and observe the current behavior of a running system and then check to see if it is consistent with the behaviors allowed by the model.

In this position paper, we briefly overview our approach to the monitoring and checking at run-time and then describe issues and extensions that need to be addressed and developed to validate model-based embedded systems at run-time.

2 Model-based Monitoring and Checking

Although testing can be used to validate an implementation, one of the most difficult tasks during testing is to come up with a judicious collection of test cases, which probe all the different components of embedded system software. This is especially difficult when it comes to embedded systems, which need to interact with their environment within timing constraints. In particular, because of the large number of possible behaviors, it is infeasible to guarantee the correctness of a system implementation on all possible input sequences, via testing. Here, the continuous monitoring of the run-time behavior of a system can improve our confidence about the system by ensuring that the current execution is consistent with its requirements at run-time.

Figure 1 shows the overall structure of the Monitoring and Checking (MaC) framework introduced in [7]. The framework includes two main phases: static phase and dynamic phase. During the static phase, i.e., before a target program is executed, run-time components such as a filter, an event recognizer, and a run-time checker are generated from a target program and a formal requirements specification. During the dynamic phase, the instrumented target program is executed while being monitored and checked with respect to the requirements specification.

The Meta-Event Definition Language (MEDL) is used to express requirements. It is based on an extension of a linear-time temporal logic. It can be used to express a large subset of safety properties of systems, including real-time properties. We use events and conditions to capture and reason about temporal behavior and data behavior of the target program execution; events are abstract representations of time and conditions are abstract representations of data. For formal semantics of events and conditions, see [8].

Requirements written in MEDL are expressed in terms of high-level events and conditions. Events occur instantaneously during execution, whereas conditions represent information that holds for a duration of time. For example, an event denoting return from method Raise Gate occurs at the instant the control returns from the method, while a condition ($\text{TrainPosition} = 2$) holds as long as the variable `TrainPosition` does not change its value from 2. The distinction between events and conditions is very important in terms of what the monitor can infer about the execution based on the information it gets from the filter. The monitor can conclude that an event does not occur at any moment except when it receives an update from the filter. By contrast, once the monitor receives a message from the filter that variable position has been assigned the value 2, we can conclude that position retains this value until the next update.

In addition to the requirements written in MEDL, a monitoring script relates these events and conditions with low-level data manipulated by the system at run-time. Monitoring scripts are

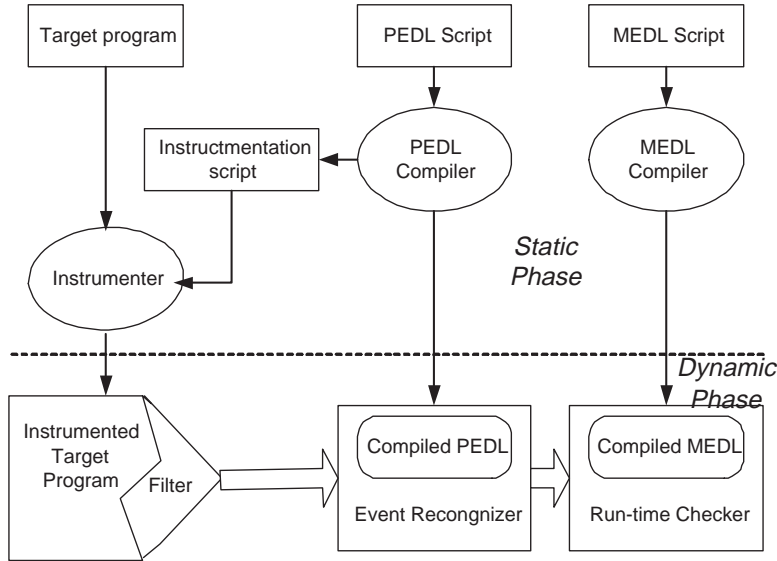


Figure 1: The MaC framework

expressed in the Primitive Event Definition Language (PEDL). PEDL describes primitive high-level events and conditions in terms of system objects. PEDL is used to define what information is sent from the filter to the event recognizer, and how it is transformed into events used in high-level specification by the event recognizer.

Based on the monitoring script, the system is automatically instrumented to deliver the monitored data to the event recognizer at run-time. The event recognizer, also generated from the monitoring script, transforms this low-level data into abstract events and delivers them to the run-time checker. The run-time checker verifies the sequence of abstract events with respect to the requirements specification and detects violations of the requirements.

The reason for keeping the monitoring script (PEDL) distinct from the requirements specification (MEDL) is to maintain a clean separation between the system itself, implemented in a certain way, and high-level system requirements, independent of a particular implementation. PEDL, therefore, is tied to the implementation language of the monitored system in the use of object names and types. MEDL is independent of the monitored system. The separation between PEDL and MEDL ensures that the architecture is portable to different implementation languages and specification formalisms. Note that implementation-dependent event recognition insulates the requirement checker from the low-level details of the system implementation. This separation also allows us to perform monitoring of heterogeneous distributed systems. A separate event recognizer may be supplied for each module in such system. Each event recognizer may process the low-level data in a different way, and all deliver high-level events to the checker in a uniform fashion.

A filter is a collection of probes inserted into the target program. The essential functionality of a filter is to keep track of changes of monitored objects and send pertinent state information to the event recognizer. An event recognizer detects an event from the state information received from the filter. Events are recognized according to a low-level specification. Recognized events are sent to the run-time checker. Although it is conceivable to combine the event recognizer with the filter, we chose to separate them to provide flexibility in an implementation of the architecture. A run-time checker determines whether or not the current execution history satisfies a requirement specification. The execution history is captured from a sequence of events sent by the event recognizer.

Meta Event Definition Language (MEDL). The safety requirements (invariants) are written in MEDL. A MEDL specification includes the following sections:

Imported events and conditions. A list of events and conditions delivered by the event recognizer is declared. Definitions of imported events and conditions are given in a separate PEDL script.

Definitions of events and conditions. Events and conditions are defined using imported events, imported conditions, and auxiliary variables, whose role is explained later in this section. These events and conditions are then used to define safety properties and alarms.

Safety properties and alarms. The correctness of the system is described in terms of safety properties and alarms. Safety properties are conditions that must be always true during the execution. Alarms, on the other hand, are events that must never be raised. Alarms and safety properties are complementary ways of expressing the same thing. The reason that we have both of them is because some properties are easier to think of in terms of conditions, while others are in terms of alarms.

Auxiliary variables. In order to increase expressive power of MEDL, we allow auxiliary variables to be used in MEDL specifications. Auxiliary variables can be used to define events and conditions, and their values are updated in response to events. Auxiliary variables allow us, for example, to count the number of occurrences of an event, or refer to the *i*th occurrence of an event. For example, the variable `count_e1`, updated by the occurrences of the event `e1` as follows:

```
    e1 -> count_e1' := count_e1 + 1;
```

counts occurrences of event `e1`. The variables `count_e1'` and `count_e1` denote the new and old values of the variable `count_e1`. A special auxiliary variable `currentTime` is used to refer to the current time of the target program. It is set to be the last timestamp received from the filter.

Primitive Event Definition Language (PEDL). PEDL is the language for writing low-level specifications that map run-time data of the system execution into high-level events and conditions. PEDL encapsulates all implementation-specific aspects of the monitoring process, and therefore, is by necessity specific to the target programming language. A PEDL specification consists of the following sections.

Exported events and conditions. Events and conditions to be sent from an event recognizer to a run-time checker are declared, and must match the imported section of the MEDL script.

Declarations of monitored entities. This section is specific to the target programming language. In a typical programming language, monitored entities are variables and control locations in the program.

Definitions of events and conditions. This section uses monitored entities declared in the previous section to define events and conditions. Again, this is specific to the target programming language. Typical primitive events are updates of monitored variables and visits to monitored control locations, such as function calls. In order to make event recognition fast, we require that all events and conditions defined in a PEDL specification are in terms of a single state in the execution trace. That is, no history information needs to be stored by the event recognizer.

Java-MaC. To demonstrate the effectiveness of the MaC framework, we have implemented a MaC prototype for Java programs, called Java-MaC. Java-MaC targets Java executable code (i.e., bytecode), where actual modifications to the bytecode are carried by `jtrek`. It is easy to deploy Java-MaC, because it automatically instruments the target program and generates the run-time

components of Java-MaC based on requirements specifications written in two scripting languages, MEDL and PEDL-for-Java. The system is available at www.cis.upenn.edu/~rtg/mac.

3 Position Statement

We believe that the run-time monitoring and checking approach is necessary to ensure the correct operation of real-time and embedded systems. Our MaC framework seems quite general as shown in different application domains[5, 2]. Here, we describe issues and extensions needed to apply the MaC architecture to embedded systems. They are to check properties based on hybrid systems, to develop fundamental understanding of distributed property checking, to add synchronous steering capability, and to develop model-based profiling as follows.

MaC based on Hybrid System Models. Embedded systems are often designed to interact with complex physical processes, where some parameters of the system change continuously. To model such systems, the various *hybrid systems* modeling languages have been developed; e.g., CHARON [1] which supports the modular and hierarchical modeling of hybrid reactive systems. The run-time monitoring and checking becomes more complex when the system behavior involves continuous changes to the system state, as well as discrete changes. We need to understand sampling based monitoring of property and also to be able to reason about errors due to sampling at run-time.

Distributed MaC. This direction deals with the increased role played by distributed embedded systems. Using a centralized monitoring approach for distributed system will lead to a bottleneck. It is important to develop an approach to distributed property checking to allow us to have local monitors and checkers that reside together with each node. They should collectively ensure that the system execution is correct while minimizing the communication overhead of the monitoring layer. It is not how to decompose a global property into local properties that can be checked efficiently.

The problem of distributed MaC requires fundamental understanding and needs new concepts that are not currently available under the notion of run-time verification. The problem can be stated as follows: Given a set of nodes, N_1, \dots, N_m and a global property P_g , find locally checkable properties, P_1, \dots, P_m such that if each N_i satisfies its property P_i , then the system as a whole satisfies P_g . In MaC, we want to ensure that if node N_i raises an alarm because its property P_i is violated, then it ensures that the global property P_g is violated. The issues are how to decompose P_g into P_1, \dots, P_m and evaluate the state information that needs to be exchanged between nodes. In the worst case, each node can have the same information as any other node. Obviously, if every node maintains its copy of the global state, then whenever it detects a failure or raises an alarm, the system as a whole has failed. But this is not a solution that scales.

Steering-based Adaptation. Since it is not good enough to tell something bad is about to happen, the monitoring and checking layer should allow feedback to the monitored system. The feedback should then be used by the system to recover from the problems discovered by the checker and either reconfigure itself or take a corrective action. We call this feedback steering. We have been experimenting with extending the MaC system with some form of steering; however, it lacks flexibility and a firm theoretical basis. While monitoring and checking has well-founded theoretical basis in terms of temporal logics and value abstractions [10], the current implementation of steering is rather *ad hoc*. We need to have a better understanding of how steering can be applied in a safe manner, what kinds of formal guarantees can be provided by the use of steering, and what kinds of applications will benefit most from the use of steering.

Model-based Profiling. One area that has not been explored is what we call model-based profiling. The use of run-time verification technology such as MaC guarantees that the current execution

if the system is correct with respect to the set of properties that represent system requirements. However, the run-time data from prior runs are not saved. We believe that such information can be used to establish a dynamic measure of system dependability. The metric is based on the degree of coverage computed from information collected during run-time monitoring and is refined over the life-time of the system. Intuitively, the longer the system runs problem-free, the more likely it is to behave well the next time it is used; that is, models that are statistically validated at run-time. Such measures can be quite useful when checking interoperability of components. Instead of checking for all possible behaviors, this will allow checking for the behaviors that are likely to be used.

There are two ways to acquire models used for model-based profiling. One approach is use design specifications of the system. Here, we can in addition to computing the dependability measure, ensure that the system indeed implements the design specification, complementing the property validation by the checker. Another approach is to use existing model extraction tools, such as Java PathFinder [4].

4 Summary

We have developed the two-level architecture for run-time monitoring and checking, called MaC. The separation between MEDL and PEDL ensure that the architecture is portable to different implementation languages and specification formalisms; e.g., MEDL is to be platform independent and PEDL is to be platform specific¹. This paper identifies several tasks and extensions that would make the MaC architecture useful for distributed real-time and embedded systems.

References

- [1] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specifications of hybrid systems in CHARON. In *Proceedings of Hybrid Systems: Computation and Control, Third International Workshop*, volume 1790 of *LNCS*, pages 6–19. Springer-Verlag, 2000.
- [2] Karthikeyan Bhargavan, Carl A. Gunter, Moonjoo Kim, Insup Lee, Davor Obradovic, Oleg Sokolsky, and Mahesh Viswanathan. Verisim: Formal analysis of network simulations. In *Proceedings of International Symposium on Software Testing and Analysis*, August 2000.
- [3] Aniruddha Gokhale, Douglas C. Schmidt, Balachandra Natarajan, and Nanbor Wang. Applying Model-Integrated Computing to Component Middleware and Enterprise Applications. *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules*, 45(10), October 2002.
- [4] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000.
- [5] Klaus Havelund and Grigore Rosu. Synthesizing Monitors for Safety Properties. In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS2002)*, April 2002.
- [6] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS2002)*, April 2002.

¹Similar separations between platform specific and platform independent models have been proposed and advocated by Model Driven Architecture (MDA) by OMG and Model Integrated Computing (MIC) [9, 3].

- [7] Moonjoo Kim, Mahesh Viswanathan, Hanène Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *Proceedings of the European Conference on Real-Time Systems (ECRTS '99)*, pages 114–121, June 1999.
- [8] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
- [9] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, April 1997.
- [10] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, CIS Dept. Univ. of Pennsylvania, 2000.