Steering of Real-Time Systems Based on Monitoring and Checking*

Oleg Sokolsky, Sampath Kannan, Moonjoo Kim, Insup Lee, and Mahesh Viswanathan
Department of Computer and Information Science
University of Pennsylvania
{sokolsky,kannan,moonjoo,lee,maheshv}@saul.cis.upenn.edu

Abstract

We present an approach to enhance fault-tolerance of real-time systems through steering. Steering means external alteration of the system's behavior in response to a deviation from requirements. The steering technique is embedded into a framework of monitoring and checking (MaC). MaC allows the users to perform run-time analysis of the current execution of a system with respect to formally specified requirements. We describe our current and future work on steering, including the language to specify steering actions and system instrumentation that enables steering. A prototype implementation for monitoring and steering of Java programs is also presented.

1 Introduction

This paper addresses the problem of run-time correctness of a computer system. Current state of the art in system analysis and verification does not guarantee that the execution of a system will comply with its requirements. On the one hand, formal verification analyzes all executions of the system, but the analysis is performed on the specification of the system, not its implementation. In addition, state-of-the-art verification techniques still do not scale up to large real systems. On the other hand, testing performs analysis of the system implementation, but does not guarantee that all behaviors are analyzed. Whatever approach is used to analyze the system before its deployment, it is possible that the system will misbehave at run time.

Our approach to alleviate this problem is to detect deviations of the system behavior from its requirements, and correct it by means of external intervention. The goal is to let the system continue with its execution without resorting to a full reset. Instead, the run-time data of the system are adjusted in order to eventually restore the system to a correct execution. We call this on-line correction *steering*. The property provided by the steering component should be that after a period of incorrect behavior the system is always restored to normal operation.

The design philosophy of our steering approach is that the system has been analyzed before execution. Therefore, its behavior is mostly correct, except maybe for a few subtle cases. Therefore, the steering component does not try to take over the control algorithm of the system. Instead, it attempts to help the system recover from the detected violation by tuning parameters of the algorithm or by adjusting the state of one of the system components.

An example of a problem that can be corrected by steering is given by a system of two processes communicating through a faulty medium. Communication protocols are designed with certain assumptions about the behavior of the medium. An unlikely combination of faults may violate these assumptions and cause the processes to lose synchronization and spend their time in useless retransmissions. The conflict can be resolved by steering. Possible steering actions in this situation are to cause the receiver to drop some of the messages from the incoming queue or to force the sender to produce a modified message that will be accepted by the receiver.

There are several steps that need to be carried out by the implementation of steering: 1) Requirements to system executions need to be stated in such a way as to permit run-time detection of violations, giving enough information to diagnose the source of the problem. 2) The system needs to be monitored in order to collect the data for the detection and diagnosis. 3) Appropriate corrective measures need to be identified based on the diagnostic data. 4) Correction has to be introduced into the running system in such a way that would not interfere with the execution.

*This research was supported in part by ONR N00014-97-1-0505 (MURI), NSF CCR-9619910, ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466

Monitoring, needed in step 2), is an important part of steering. To leverage research and development effort with the results of prior work, we are implementing steering within the monitoring and checking (MaC) framework [6]. The MaC framework has been designed to test run-time compliance of an execution of a real-time system with its formal requirements. MaC operates by extracting low-level run-time data from the system execution and converting them into a stream of high-level events. The event stream is checked for compliance with requirements expressed in a formal language. Requirement violations are reported to the user.

The MaC framework provides us with a tool to solve tasks 1) and 2) outlined above. In this paper, we discuss the issues concerning the other two steering tasks and describe their implementation by means of an extension to MaC. The rest of the paper is organized as follows: Section 2 gives an overview of the MaC framework; Section 3 we outline the concepts underlying steering. Section 4 introduces the scripting language for steering. In Section 5 we discuss the prototype implementation of MaC with steering. We conclude with an outline of directions of ongoing and future research.

2 The Monitoring and Checking Framework

The structure of the MaC framework is demonstrated in Figure 1. The user specifies the requirements of the system in a formal language. Requirements are expressed in terms of high-level events and conditions. In addition, a *monitoring script* relates these events and conditions with low-level data manipulated by the system at run time. Based on the monitoring script, the system is *automatically* instrumented to deliver the monitored data to the *event recognizer*. 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 requirements.

The reason for keeping the monitoring script distinct from the requirements specification is to maintain a clean separation between the system itself, implemented in a certain way, and high-level system requirements, independent of a concrete implementation. 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 way specific to the respective module. For example, an event recognizer that is associated with a software component will work very differently from the one that processes traffic on a bus. But all event recognizers deliver high-level events to the checker in a uniform fashion.

In keeping with this design philosophy, two languages have been designed for use in the MaC framework. The Meta-Event Definition Language (MEDL) is used to express requirements. It is based on an extension of a linear-time temporal logic. It allows to express a large subset of safety properties of systems, including real-time properties. Monitoring scripts are expressed in the Primitive Event Definition Language (PEDL). PEDL describes primitive high-level events and conditions in terms of system objects. 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.

Run-time monitoring provides for efficient detection of violations of system requirements and raise an alarm when a violation happens. At the same time, the vast information collected during monitoring in order to detect requirement violations allows us to go one step further. The same information can be used to diagnose the problem that has lead to the violation and suggest a remedy for it. In order to apply this remedy, we need the means to provide feedback from the run-time checker back into the system. The MaC framework gives us, in addition to the run-time information about the system behavior, instrumentation facilities that can be used to automatically establish this feedback and influence the system behavior to restore compliance with the requirements. Steering provides the means to establish the feedback from the checker into the monitored system.

The work to incorporate steering into the MaC framework consists of the following steps: (1) requirements for the steering module have been formulated; (2) a scripting language for specification of actions involved in steering has been designed; (3) the Java-based MaC prototype has been extended to incorporate steering.

3 Steering Concepts

Steering a system at run time involves three necessary steps: 1) detect a problem, 2) diagnose the malfunction, and 3) invoke steering. Each stage requires a number of design decisions to be made. An important question is, how tight the integration between the system and the monitor should be? Tighter cou-

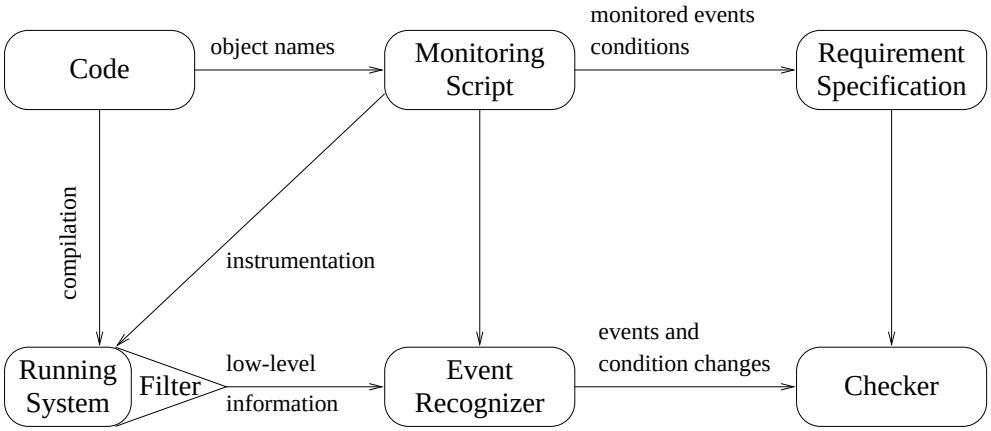

Figure 1. MaC framework

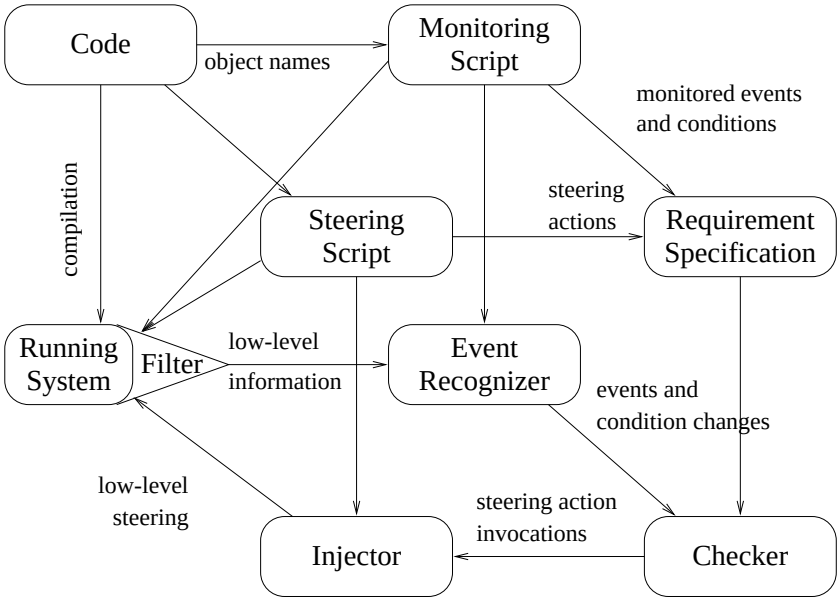

Figure 2. The monitoring and steering framework

pling makes communication between the system and the monitor more efficient and thus decreases steering latency, that is, the time between the occurrence of a malfunction and the steering action to correct it. On the other hand, tight coupling means that the system and the monitor will compete for the same computational resources, potentially inhibiting the system performance. In our approach, we assume that the system has limited computational resources and cannot accommodate a local monitor. Moreover, we assume that the system under consideration has been already designed, implemented and deployed, so it is impossible to incorporate the monitor into the system itself.

Detection of requirement violations is accomplished by monitoring events used in the requirement specification. In addition, additional data need to be monitored for diagnostic purposes. For example, if a receive event did not follow the send event soon enough, it may either mean that the other party did not respond or that the medium failed to propagate the response, and different actions need to be taken.

The main vehicle for steering is a *steering action*. Steering actions are specified by the user. Steering actions are invoked by the monitor after requirement violations. Action invocations are transmitted to the system and "injected" into the system execution. Actions are then executed locally within the system, and have access to all objects of the system.

System objects that are modified by steering actions are called *steered objects*. A steering action can modify any variable inside any object that the user can specify unambiguously in the steering script (discussed in the next section). There are several ways to assign a new value to the steered variable. A value may be computed statically, so that a constant is assigned to the variable at run time. This is the most efficient and the least powerful way. Alternatively, the value may be computed at run time by the monitor. The drawback of the monitor-side computations is that it may require to transfer system objects used in the intermediate computations that are not necessary for detection and diagnosis. Additional transfer of data creates excessive communication load between the system and the monitor, potentially increasing steering latency. The most efficient way may to perform the run-time computations on the system side, provided they do not require inordinate amounts of CPU time, by calling a procedure already existing in the system code. To provide for this, a steering action may invoke methods of a steered object. In addition to reducing the system/monitor communication, system-side computation saves the user from repeating in the steering script the computation that may already be provided by a method in the steered object. From a pure object-oriented point of view, it may be cleaner to restrict steering actions to method calls. This would guarantee that integrity of every steered object is preserved by all steering actions. However, an object may have an overly restrictive interface that will not allow an action to perform steering. Finally, the monitor can introduce, through instrumentation, additional code that is executed within the system in response to steering action invocations. Although the most powerful, this last method is also the most dangerous and should be used with caution.

The basic underlying premise for steering in our approach is that steering will be needed infrequently. Because of this, it is important that the framework satisfies the *conservativeness* property. That is, the system does not suffer a significant performance degradation as long as steering is not invoked.

Another important requirement is *effectiveness*. It ensures that a steering action cannot be ignored by the system. One aspect of this requirement is that the steering component must be able to change every object that it needs to change in order to perform the action. Another, more subtle, aspect is that the system should not be able to "undo" the effect of a steering action. This can happen if the system modifies an object shortly after it has been modified by a steering action. Because of this, the steering framework must ensure that an action is invoked when it will be effective, delaying its execution if necessary. There is another reason why steering actions cannot be allowed to be invoked at arbitrary moments. Execution of a steering action can interfere with the system execution and cause harm instead of correcting the system's behavior.

Steering scripts, give the user the ability to control the moment when a steering action is executed, thereby ensuring its effectiveness. This is done by means of steering conditions associated with each action. Execution of a steering action is delayed until its condition is satisfied. Steering conditions can be either static or dynamic. Static conditions are fully evaluated during instrumentation, while dynamic conditions depend on run-time information. Dynamic conditions provide for finer control of action invocations. On the other hand, additional effort to evaluate the conditions at run time can compromise conservativeness of steering. In the current prototype, only static conditions are implemented.

Implementation of steering actions may be different depending on the target system and the goals of steering. Two approaches are possible: *threaded* and *procedural* actions. Threaded steering actions are exe-

cuted concurrently with the system activity in a separate thread, while procedural actions are invoked by a system thread at specific points during the execution, determined by instrumentation. Each method has its advantages and limitations.

The procedural approach can use only static steering conditions since locations for the steering actions must be decided before the execution. When steering conditions yield large regions of code where steering is allowed, a procedural implementation may be inefficient, either because it may delay a steering action unnecessarily, or because it checks for action invocations multiple times within the steering region. On the other hand, since the steering action is executed within the system thread, there is no need for steering synchronization and all objects accessible to that thread can be used in steering actions.

By contrast, threaded steering can utilize both static and dynamic steering conditions, and can invoke a steering action as soon as its condition is satisfied. However, synchronization between the system thread and the steering thread is required, bringing additional overhead. When steering conditions are tight, that is, steering is allowed only for short intervals, this overhead can be very high. Moreover, certain objects that are local to the system thread may be inaccessible to the steering thread. For example, when steering Java programs, a separate steering thread cannot affect local variables of a method. These concerns led us to implement procedural steering in our current prototype.

The structure of the MaC framework extended with steering is shown in Figure 2. The steering script is provided by the user in addition to the monitoring script. The steering script describes steering actions and conditions for their invocation. Like the monitoring script, the steering script deals with the low-level information about system objects and therefore depends on the system implementation. The requirement specification now additionally has to specify which steering action is triggered by a requirement violation. Only the invocation of a steering action is described, based on action names exported by the steering script. Therefore, the requirement specification is still independent of the system implementation.

The functionality of the run-time checker is extended with the ability to invoke steering actions and pass them to a new run-time component called the injector, which is incorporated into the running system during instrumentation. The injector accepts invocations of steering actions and translates them into low-level steering of the objects of the system according to the steering script.

```
steering script mav

steered objects
  Air    MAV:air;
  Point  MAV:position;

steering action controlRepulsion( boolean tf ) =
  { call (MAV:air).setRepulse(tf); }
          before write MAV:position;

end
```

Figure 3. A sample steering script

4 A Language for Steering Actions

To specify steering actions, we designed a special scripting language SADL (**S**teering **A**ction **D**efinition **L**anguage). The steering scripts written in SADL specify how the system objects are affected by a steering action. Figure 3 shows a sample script, taken from a study in steering of artificial physics algorithms [3]. In the example, a pattern of particles is being formed by applying forces of attraction and repulsion between the particles. If a problem is discovered, the checker steers the system by manipulating the force of repulsion.

The script consists of two main sections: declaration of steered objects (that is, system objects that are involved in steering) and definition of steering actions where the declared objects are used. Since steering is performed directly on the system objects, SADL scripts are by necessity dependent on the implementation language of the target system. Our prototype implementation of the monitoring and steering framework (described in the next section) aims at systems implemented in Java. Because of this, SADL scripts used in the prototype are also tied to Java.

The first section of the script defines steered objects. Objects used to express steering conditions are also declared in this section. The steered objects can be fields and methods of Java classes as well as local variables of methods. In the example, steered objects are the `Air` object, a repository of the algorithm parameters shared by all particles, which is used by the steering action, and the variable representing the position of a particle, used in the steering condition.

The second section of the steering script defines steering actions and specifies steering conditions. An action can have a set of parameters that are computed by the checker and passed to the system together with the action invocation. The body of an action is a collection of statements, each of which is either a call to

```
ReqSpec mav

import action controlRepulsion(boolean);

alarm noPattern = ...;

noPattern -> { invoke controlRepulsion(true); }

end
```

Figure 4. Action invocation in the MEDL script

a method of the system or an assignment to a system variable. In the example, the steering action calls a method that controls repulsion between particles, and is allowed to happen every time the position of a particle is about to be updated.

In addition to a steering script, the requirement specification language is extended to provide for invocation of steering actions. An action is invoked in response to an occurrence of an event or an alarm. Figure 4 presents a fragment of the MEDL script of the artificial physics example. It shows the declaration of the steering action `controlRepulsion`, imported from the steering script, and the alarm `noPattern` that is raised by the checker when it detects a violation of the pattern formation. The definition of the alarm is rather complex and is omitted for clarity. When the alarm is raised, the steering action is invoked with the `true` value of its parameter, which suspends repulsion between particles and triggers the process of restoring the pattern.

5 Steering in the MaC Prototype

A prototype implementation of the MaC framework has been implemented and tested on a number of examples. The prototype is targeted towards monitoring and checking of programs implemented in Java. Java has been chosen as the target implementation language because of the rich symbolic information that is contained in Java class files, the executable format of Java programs. This information allows us to perform the required instrumentation easily and concentrate on the more fundamental aspects of the monitoring and checking framework implementation. Figure 5 shows the design of the Java-based MaC prototype.

The PEDL language of the prototype allows the user to define primitive events in terms of the objects of a Java program: updates of program variables (fields of a class or local variables of a method) and method calls. Automatic instrumentation guarantees that all relevant updates are detected and propagated to the event recognizer.

The prototype uses interpreters for PEDL and MEDL. Each interpreter includes a parser for the respective language and works on a parsed version (the abstract syntax tree) of a script. The MEDL interpreter is the run-time checker. It accepts primitive events sent by the event recognizer and re-evaluates all events and condition described in the MEDL script. The PEDL interpreter is the event recognizer. It accepts the low-level data sent by the instrumented program and, based on the definitions in the monitoring script, detects occurrence of the primitive events and delivers them to the run-time checker. In addition, the PEDL interpreter produces the instrumentation data that is used to automatically instrument the system.

The MaC instrumentor is based on JTREK class library [5], which provides facilities to explore a Java class file and insert pieces of bytecode, preserving integrity of the class. During instrumentation, the instrumentor detects updates to monitored variables and calls to monitored methods and inserts code to send a message to the event recognizer. The message contains the name of the called method and its parameter values, or the name of the updated variable and its new value. Each message contains a time stamp that can be used in checking of real-time properties.

In order to implement steering in the MaC prototype [6], we had to add the following components:

- A parser for SADL. The parser produces two components: 1) a list of actions together with their conditions in the form that can be used by the instrumentor; 2) a new class, `Injector`, discussed below.
- The injector is the component responsible for communication with the checker. When the system is started, the injector is loaded into the virtual machine of the monitored system. At run time, when a steering action happens, the injector receives a message from the checker and sets a flag to indicate that the steering action has happened. The bodies of the steering actions are also represented in the prototype as methods of the `Injector` class.
- The functionality of the instrumentor is extended to insert the additional code at the positions prescribed by the steering conditions. The code tests the flag for action invocations and makes calls to the injector to execute the action.
- The run-time checker is extended to handle action

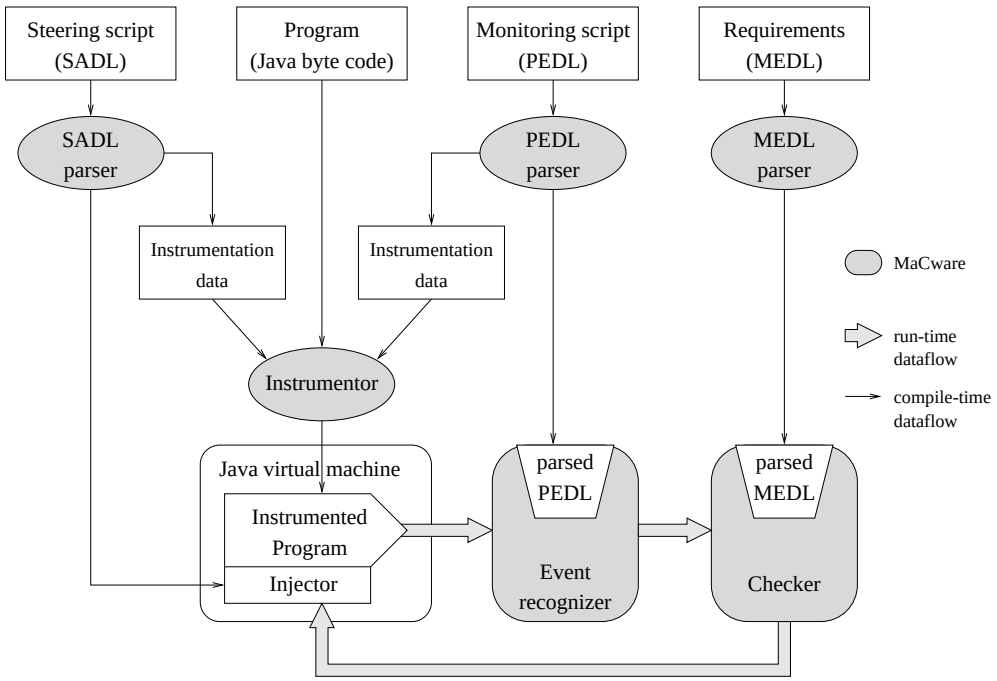

Figure 5. Java-based MaC prototype

invocations. A connection between the checker and the injector is established at system start-up. The MEDL interpreter processes invocation instructions and sends the corresponding messages to the injector.

6 Conclusions

We have described an approach to perform run-time correction of system behavior by means of steering actions. Steering is embedded into a monitoring and checking framework, which detects violations of formal requirements in the observed execution of the monitored system. Monitoring, in conjunction with steering, provides an additional layer of fault tolerance in the system.

Related work. Program steering has been extensively explored in the context of performance optimization and resource management. Systems such as Falcon [4], MOSS [1], Autopilot [7] aim at steering of large distributed systems. Most of these systems rely on interactive steering, where decisions are made by a human user, although automatic control is also possible. MaC, on the other hand, emphasizes automatic procedures both during instrumentation and at run time. Another significant distinction of MaC is that it targets correctness of the system execution rather than its performance, and relies on formal specification of correct behaviors.

MaC is also distinguished from other approaches to fault tolerance (see, for example, [2]). Although the goals and some of the methods are similar, we are not trying to provide an alternative to fault-tolerant components within the system itself. It is clear that such components are likely to be more efficient than an external layer such as MaC. Rather, MaC should be used when the system requirements change after the system has been designed, implemented and deployed, or the environment is more demanding than anticipated at design time. In this case, incorporating additional fault tolerance into the system itself may be impossible and an external solution is justified.

Current and future research. The work on the steering framework is actively under way. Our current goal is to achieve a better understanding of the theoretical basis for steering. Questions that need to be answered include:

- what problems can be resolved by means of steering? Clearly, we cannot reverse catastrophic failures. However, faults can be detected and corrected before they develop into a failure. We are developing a fault model that will characterize faults amenable to correction by steering.

- what is the right way to reason about steering? We need to perform high-level analysis of effects of a particular steering action. In order to do this, we are working on a high-level state-machine model of the steering component. Together with the fault model, a precise behavioral description of steering will allow users to perform analysis of the system together with steering.

The current prototype will serve as an important vehicle in exploring the possibilities and shortcomings of steering. Several case studies in monitoring and steering of systems is under way, and we expect to gain much experience from them.

References

[1] G. Eisenhauer and K. Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, Aug. 1998.

[2] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in distrbuted environments. *ACM Computing Surveys*, 31(1), Mar. 1999.

[3] D. Gordon, W. Spears, O. Sokolsky, and I. Lee. Distributed spatial control and global monitoring of mobile agents. In *Proceedings of the IEEE International Conference on Information, Intelligence, and Systems - ICIIS'99, to appear*, Nov. 1999.

[4] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, Aug. 1998.

[5] Java Technology Center, Compaq Corp. *Compaq JTrek* . Online documentation: http://www.digital.com/java/download/jtrek/.

[6] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Proceedings of the European Conference on Real-Time Systems - ECRTS'99*, pages 114–121, June 1999.

[7] R. Ribler, H. Simitci, and D. Reed. The *Autopilot* performance-directed adaptive control system. *Future Generation Computer Systems*, 1999. To appear.