Run-time Monitoring and Steering based on Formal Specifications*

Sampath Kannan, Moonjoo Kim, Insup Lee†,
Oleg Sokolsky, and Mahesh Viswanathan
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, U.S.A

August 30, 2000

Abstract

We describe the Monitoring-aided Checking and Steering (MaCS) framework that assures the correctness of software execution at run-time. Checking is performed based on a formal specification of system requirements to ensure that the current system behavior is in compliance with these requirements. When the system behavior violates these requirements, steering is invoked to correct the system. Our framework bridges the gap between formal verification and testing. The former is used to ensure the correctness of a design specification rather than an implementation, whereas the latter is used to validate an implementation. The paper presents an overview of the framework and the three scripting languages, which are used to specify what to observe from the running program, the requirements that the program should satisfy, and how to steer the running program to a safe state. An important aspect of the framework is clear separation between the implementation-dependent description of monitored objects and the high-level requirements specification. Another salient feature is automatic instrumentation of executable code for monitoring and steering. This paper also describes our current prototype implementation in Java.

1 Introduction

The design analysis and verification of distributed and real-time systems has become an important research topic over the past two decades. Important results have been achieved, in particular, in the area of formal verification [4]. Formal methods of system analysis allow developers to specify their systems using mathematical formalisms and prove properties of these specifications. These formal proofs increase confidence in correctness of the system's behavior. Complete formal verification, however, has not yet become a practical method of analysis. The reasons for this are twofold. First, the complete verification of real-life systems remains infeasible. The growth of software size and complexity seems to exceed advances in verification technology. Second, verification results apply not to system implementations, but to formal models of these systems. That is, even if a design has been formally verified, it still does not ensure the correctness of a particular implementation of the design. This is because an implementation often is much more detailed, and also may not strictly follow the design.

One way that people have traditionally tried to overcome this gap between design and implementation has been to test an implementation on a predetermined set of input sequences. This approach, however, fails to provide guarantees about the correctness of the implementation since not all possible behaviors can be tested. For mobile code, testing may not even be possible, especially if such code is downloaded on demand for execution. Conse-

*This research was supported in part by ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, NSF CCR-9619910, NSF CCR-9988409, and ONR N00014-97-1-0505 (MURI).

†POC: lee@cis.upenn.edu

quently, when the system is running, it is hard to guarantee whether or not the system is executing correctly.

Computer systems are often monitored for performance measurement, evaluation and enhancement as well as to help debugging and testing [24]. Lately, there has been increasing attention from the research community to the problem of designing monitors that can be used to assure the correctness of a system at runtime [1, 5, 23, 19, 22, 17, 15]. These systems, however, tend to be based on informal specifications, require manual instrumentation, or depend much on the specificity of target systems. Our goal is to develop the monitoring, checking and steering framework based on formal specifications, which supports automatic instrumentation and isolates the implementation-dependency of the target system.

The overall structure of the Monitoring, Checking and Steering framework is shown in Figure 1. The user specifies the requirements of the system, which are expressed in terms of a sequence of abstract events, or trace. A *monitoring script* describes the mapping from observations to abstract events. The Monitor use this script to decide when and how to observe the system to extract abstract events needed by the checker. The Checker verifies the sequence of abstract events with respect to the requirements specification, detects violations of requirements and generate a meta-event as the result. The Steerer uses the sequence of meta-events to decide how to adjust the system dynamically to a safe state through control events.

In the next section, we decribe the framework. In keeping with the design philosophy of the framework, we have developed three languages in our prototype implementation. The Meta-Event Definition Language (MEDL) is used to express requirements. MEDL is based on an extension of a linear-time temporal logic. It allows us to express a large subset of safety properties of systems, including real-time properties. MEDL is described in Section 3 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. The Steering Action Definition Language (SADL) is used to specify how a system is affected by steering actions. Section 4 describes the prototype implementation for Java as well as PEDL and SADL.

2 Overview of the Framework

The Monitoring, Checking and Steering (MaCS) framework specifies components that are necessary to perform run-time correctness monitoring of a system. It is independent of the system implementation. Of course, any concrete implementation of the framework will have to interface with the system to ensure proper exchange of information between the system and the monitor. In describing the framework, we carefully separate system-dependent components from system-independent ones. System-dependent components are presented in the context of an existing prototype implementation of the framework.

The overall structure of the MaCS framework is shown 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 (see Section 3). 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 a stream of observations to the *monitor*. Observations are low-level data such as values of variables, method calls, etc. The monitor, also generated from the monitoring script, transforms this low-level data into abstract events. Since detection of abstract events is the primary function of the monitor, we also refer to it as the *event recognizer*.

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 performed by the monitor 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

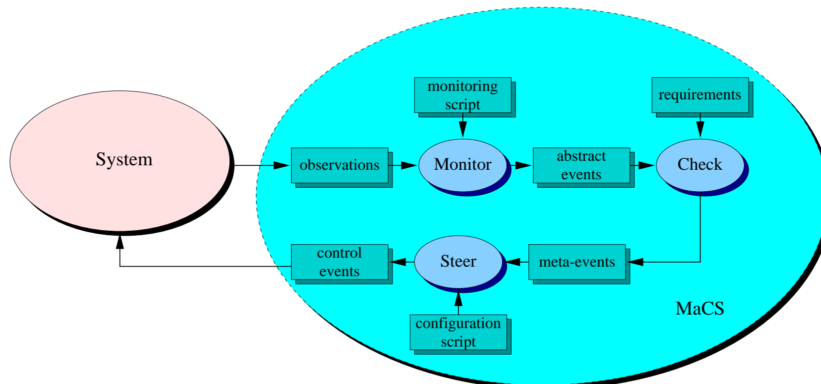

Figure 1: The Monitoring, Checking and Steering Framework

events to the checker in a uniform fashion.

The abstract events recognized by the monitor are delivered 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. When a violation is detected, the checker raises an alarm. Besides the identification of the violation, the checker may be able to provide diagnostic information to the user, based on the data collected during monitoring. Because of the rich content, the outputs of the checker are called *meta-events*.

Run-time monitoring and checking effectively detects violations of system requirements and raise an alarm when a violation happens. The question is what to do when such a violation is detected, especially for those systems that cannot be reset and restarted. For such systems the run-time state can be adjusted to *steer* the system to a safe state through feedback from the checker to the monitored system. The design philosophy of steering follows the general idea of the framework, namely, that the system is mostly correct, except for a few subtle cases. Therefore, the steerer should not try to take over the control of the system, but help the system to recover from the detected violation by tuning parameters of the system. This approach captures the limitations of control that can be performed by a loosely coupled component such as the checker.

3 Logic for Events and Conditions

The framework provides an architecture for analyzing systems formally and flexibly using runtime information. In order to specify safety properties that are being ensured, we distinguish observations into events and conditions as in SCR [10]. *Events* occur instantaneously during the system execution, whereas *conditions* are predicates that hold for a duration of time. 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 checker assumes that truth values of all conditions remains unchanged between updates from the monitor. For events, the checker makes the dual assumption, namely, that no events (of interest) happen between updates.

Since events occur instantaneously, we can assign to each event the time of its occurrence. Timestamps of events allow us to reason about timing properties of monitored systems. A condition, on the other hand, has *duration*, an interval of time when the condition is satisfied. There is a close connection between events and conditions: the start and end of a condition's interval are events, and the interval between any two events can be treated as a condition. This relationship is made precise in the logic [14].

Based on this distinction between events and conditions, we have a simple two-sorted logic. The syntax of conditions (C) and events (E) is as fol-

lows:

```
<C> ::=   c
        | [ <E>, <E> )
        | ! <C>
        | <C> && <C>
        | <C> || <C>
        | <C> => <C>
<G> ::= <E> -> <Statements>
<E> ::=  e
        | start(<C>)
        | end(<C>)
        | <E> && <E>
        | <E> || <E>
        | <E> when <C>
```

Here `e` refers to primitive events that are reported in the trace by the monitor; `c` is either a primitive condition reported in the trace or a boolean condition defined on the auxiliary variables. Guards (G) are used to update auxiliary variables that may record something about the history of the execution.

The models for this logic are similar to those for linear temporal logic, in that they are sequences of worlds. The worlds correspond to instants in time at which we have information about the truth values of primitive conditions and events. Each world is, therefore, labeled by the time instant it corresponds to and the set of primitive conditions and events that are true at that instant. Intuitively, these worlds correspond to the times when the monitor adds something to the trace.

The intuition in describing the semantics of events and conditions based on such models, is that conditions retain their truth values in the duration between two worlds, while events are present only at the instants corresponding to certain worlds. The labels on the worlds give the truth values of primitive conditions and events. The semantics for negation (`!c`), conjunction (`c1 && c2`), disjunction (`c1 || c2`) and implication (`c1 => c2`) of conditions is defined naturally; so `!c` is true when `c` is false, `c1 && c2` is true only when both `c1` and `c2` are true, `c1 || c2` is true when either `c1` or `c2` is true, and `c1 => c2` is true if `c2` is true whenever `c1` is true. Conjunction (`e1 && e2`) and disjunction (`e1 || e2`) on events is defined similarly. Now, since conditions are true from some time until just before the instant when they become false, two events can naturally be associated with a condition, namely the instant when the condition becomes true (`start(c)`) and the instant when the condition becomes false (`end(c)`). Any pair of events define an interval of time, and forms a condition `[e1, e2)` that is true from event `e1` until `e2`. The event `e when c` is true if `e` occurs and condition `c` is true at that time instant. Finally, a guard `e -> stmt`, is executed when event `e` is true; the effect of the execution is to update the values of the auxiliary variables according to the assignments given in `stmt`. The formal semantics for this logic is given in [13, 14].

Notice that some natural equivalences hold in this logic. For example, for any condition `c`, `c = [ start(c), end(c) )`. This allows one to identify conditions with pairs of events. Also, for conditions `c1` and `c2`, and event `e`, `e when c1 when c2 = e when c1 && c2`.

3.1 Meta Event Definition Language (MEDL)

The safety requirements that need to be monitored are written in a language called MEDL. Like PEDL, MEDL is also based on the logic for events and conditions. Primitive events and conditions in MEDL scripts are imported from PEDL monitoring scripts; hence the language has the adjective "meta".

Auxiliary Variables. The logic described earlier has limited expressive power. For example, one cannot count the number of occurrences of an event, or talk about the ith occurrence of an event. For this purpose, MEDL allows the user to define auxiliary variables, whose values may then be used to define events and conditions. Auxiliary variables must be of one of the basic types in Java. Updates of auxiliary variables are triggered by events. For example,

```
RaisingGate -> {t = time (RaisingGate);}
```

records the time of occurrence of event `RaisingGate` in the auxiliary variable `t`. Expression

```
e1 -> {count_e1 = count_e1 + 1;}
```

counts occurrences of event `e1`. A special auxiliary variable `currentTime` can be used to refer to the

current time of the system. It is set to be the timestamp of the last message received from the filter.

Defining events and conditions. The primitive events and conditions in MEDL are those that are defined in PEDL. Besides these, primitive conditions can also be defined by boolean expressions using the auxiliary variables. More complex events and conditions are then built up using the various connectives described in Section 3. 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 safety properties and alarms. Safety properties are conditions that must *always* be true during the execution. Alarms, on the other hand, are events that must never be raised. Note that all safety properties [16] can be described in this way. Also observe that alarms and safety properties are complementary ways of expressing the same thing. The reason we have both of them is because some properties are easier to think of in terms of conditions, while others are easier to think of in terms of alarms.

The checker, which is generated automatically from the MEDL script, evaluates the events and conditions described in the script, whenever it reads an element from the trace. The evaluation of individual events and conditions is fairly standard based on the semantics of the logic. However, there are dependencies between different events and conditions. For example, an event `e1` that is defined in terms of an auxiliary variable that is updated by event `e2`, must be evaluated after `e2` and the variable have been updated. Hence, the checker must evaluate the events and conditions in a consistent order. In our implementation we use a DAG data structure that implicitly encodes this dependency and has additional information that allows for fast evaluation of the events and conditions. Details of this algorithm can be found in [13].

Example. We illustrate the use of MEDL using a simple but representative example. The example is inspired by the railroad crossing problem, which is routinely used as an illustration of real-time formalisms [9]. The system is composed of a gate that can open and close, taking some time to do it, trains that pass through the crossing, and a controller that is responsible for closing the gate when a train approaches the crossing and opening it after it passes. The common specification approach is to assume an upper bound on the time necessary for the gate to open or close. In reality, however, mechanical malfunctions may result in unexpectedly slow operation of the gate. A timely detection of such a violation lets the train engineer stop the train before it reaches the crossing. In this example, we monitor the controller of the gate, using the requirement that the gate is down within 30 seconds after signal *CloseGate* is sent, unless signal *OpenGate* is sent before the time elapses. Precisely, we check that if there is a signal *CloseGate*, not followed by either signal *OpenGate* or completion of gate closing, is present in the execution trace, then the time elapsed since that signal is less than 30.

```
import event OpenGate, CloseGate;
import condition Gate_Down;
//Declaration of auxilliary variables
    var float lastClose;
    var float currentTime;
//Safety properties
    property GateClosing =
         [ CloseGate when !Gate_Down,
           OpenGate || start(Gate_Down) )
         => lastClose + 30 > currentTime;
//Rules for updating auxilliary variables
    CloseGate ->
         {lastClose = time(CloseGate); }
```

Figure 2: A sample MEDL script

The correctness requirement for the gate is given in the MEDL script shown in Figure 2. The time of the last occurrence of event `CloseGate` is recorded by the auxiliary variable `lastClose`. The requirement uses the events and conditions imported from the monitoring script and states that if there was a `CloseGate` event at the time when the gate was not down, which was not followed by either event `OpenGate` or condition `Gate_Down` becoming true, then the time allotted for gate closing has not elapsed yet.

4 Java MaCS

A prototype of the 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 3 shows the overall structure of the Java-based MaCS 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, after each primitive event, re-evaluates all events and conditions described in the MEDL script that may be affected by this event and raises alarms if necessary. If a steering action is invoked in response to an alarm, the run-time checker sends the corresponding message to the system. 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 MaCS instrumentation is based on JTREK class library [12], 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 addition, if steering is to be performed, the instrumentor inserts 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 parser for SADL 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`, which is responsible for communication with the run-time 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.

During system start-up, the interpreters for PEDL and MEDL are run together with the system, either on the same computer or elsewhere on the network. Connections between the system and the interpreters are established during the system initialization.

We give a brief overview of the three languages, PEDL, MEDL, and SADL, used to describe what to observe in the program, the requirements the program must satisfy, and how to steer the running program, respectively. These languages are based on the logic for *events* and *conditions* described in Section 3.

4.1 Primitive Event Definition Language (PEDL)

PEDL is the language for writing monitoring scripts. The design of PEDL is based on the following two principles. First, we encapsulate all implementation-specific details of the monitoring process in PEDL scripts. Second, we want the process of event recognition to be as simple as possible. Therefore, we limit the constructs of PEDL to allow one to reason only about the current state in the execution trace. The name, PEDL, reflects the fact that the main purpose of PEDL scripts is to define primitive events of requirement specifications.

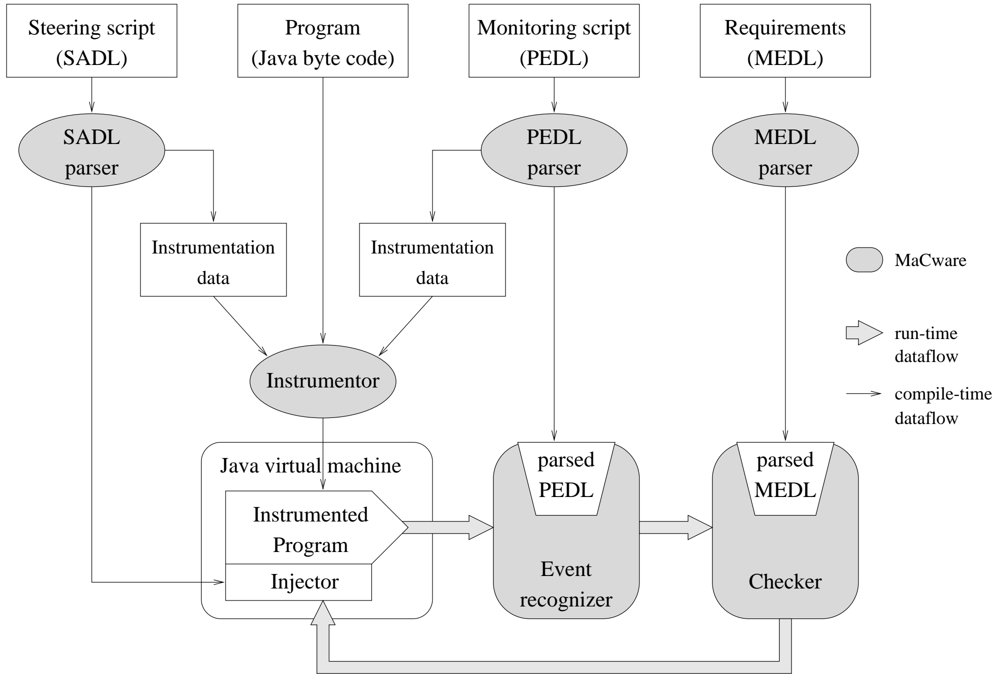

Figure 3: Java-based MaCS prototype

Monitored Entities. PEDL scripts can refer to any object of the target system. This means that declarations of monitored entities are by necessity specific to the implementation language of the system. In the current prototype which is based on Java, values of fields of an object, as well as of local variables of a method, and method calls can be monitored. Examples of monitored entities' declarations are given in Figure 5.

Defining Conditions. Primitive conditions in PEDL, are constructed from boolean-valued expressions over the monitored variables. An example of such condition is

```
condition TooFast =
     Train.Position().speed > 100.
```

In addition to these, we have primitive condition `InM(f)`. This condition is true as long as the execution is currently within method `f`. Complex conditions are built from primitive conditions using boolean connectives.

Defining Events. The primitive events in PEDL correspond to updates of monitored variables, and calls and returns of monitored methods. Each event has an associated timestamp and may have a tuple of values.

The event `update(x)` is triggered when variable x is assigned a value. The value associated with this event is the new value of `x`. Events `StartM(f)` and `EndM(f)` are triggered when control enters to and return from method `f`, respectively. The value associated with `StartM` is a tuple containing the values of all arguments. The value of an event `EndM` is a tuple that has the return value of the method, along with the values of all the formal parameters at the time control returns from the method. Besides these three, we have one other primitive event which is `IoM(f)`. This is also triggered when control returns from a method `f`, but has as its value a tuple that contains the return value of the method, and the values of the arguments *at the time of method invocation*. This event allows one to look at the input-output behavior of a method, and is needed if one wants to *program check* some numerical computation. Notice that event `IoM(f)` is the only event to violate our second design principle, namely that the operation of the event recognizer is to be based only on *the current state*.

```
class GateController {
    public static final int GATE_UP    = 0;
    public static final int GATE_DOWN  = 1;
    public static final int IN_TRANSIT = 2;
    int gatePosition;
    public void open()  { ... }
    public void close() { ... }
    ...
};
```

Figure 4: Implementation of the gate controller

All the operations on events defined in the logic can be used to construct more complex events from these primitive events. In PEDL, we also have two attributes `time` and `value`, defined for events. As mentioned in Section 3, events have associated with them attribute values, and the time of their occurrence, and these can be accessed using the attributes `time` and `value`. `time(e)` gives the time of the last occurrence of event `e`, while `value(e)` gives the value associated with `e`, provided `e` occurs. `time(e)` refers to the time on the clock of the monitored system (which may be different from the clock of the monitor) when this event occurs.

Example. Continuing with the railroad crossing example, we illustrate the use of PEDL. Figure 4 shows a fragment of the gate controller implemented as a Java class. The state of the gate is represented as variable `gatePosition`, which can assume constant values `GATE_UP`, `GATE_DOWN`, or `IN_TRANSIT`. The controller controls the gate by means of methods `open()` and `close()`. For simplicity, we assume that there is only one instance of class `GateController` in the system.

We need to observe calls to methods `open()` and `close()`, and the state of the gate. The following PEDL script introduces high-level events `OpenGate`, `CloseGate` and condition `Gate_Down`.

```
  export event OpenGate, CloseGate;
  export condition Gate_Down;
//Monitored Methods
  monmeth void GateController.open();
  monmeth void GateController.close();
//Monitored variables
  monent int  GateController.gatePosition;
//Definition of conditions
  condition Gate_Down =
       (GateController.gatePosition ==
        GateController.GATE_DOWN);
//Definition of Events
  event OpenGate =
       StartM(GateController.open());
  event CloseGate =
       StartM(GateController.close());
```

Figure 5: A sample PEDL script

4.2 A Language for Steering Actions (SADL)

Steering scripts let the user specify steering actions and control the moment when a steering action is executed, ensuring its effectiveness. This is done by means of steering conditions associated with each action. After a steering exception is raised by the checker, its execution 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 adversely affect performance of the system. In the current prototype, only static conditions are implemented.

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 6 shows a sample script, taken from a study in steering of artificial physics algorithms [8]. 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. Since the MaCS prototype implementation aims at systems implemented in Java, SADL scripts used in the prototype are also tied to Java.

```
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 6: A sample steering script

Steering entities. The entities involved in steering can be fields and methods of Java classes as well as local variables of methods. In the example, the steered entity is the `Air` object, a repository of the algorithm parameters shared by all particles. In addition, the variable representing position of a particle is used in the specification of the steering action.

Defining steering actions. 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 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.

Invocation of steering actions in MEDL scripts. 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 7 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.

```
ReqSpec mav

import action controlRepulsion(boolean);

alarm noPattern = ...;

noPattern -> { invoke controlRepulsion(true); }

end
```

Figure 7: Action invocation in the MEDL script

5 Related Work

The "behavioral abstraction" approach to monitoring was pioneered by Bates and Wileden [1]. Although their approach lacked formal foundation, it provided an impetus for future developments. Several other approaches pursue goals that are similar to ours. The work of [5] addresses monitoring of a distributed bus-based system, based on a Petri Net specification. Since only the bus activity is monitored, there is no need for instrumentation of the system. [20] generates a monitor for real-time reactive system based on a tabular requirement specification. The monitor watches over a pair of input and output of the system. The authors of [23] also consider only input/output behavior of the system. In our opinion, the instrumentation of key points in the system allows us to detect violations faster and more reliably, without sacrificing too much performance. The test automation approach of [19] is also targeted towards monitoring of black-box systems without resorting to instrumentation. In contrast, we aim at using the MaCS framework beyond testing, during real system executions. Sankar and Mandel have developed a methodology to continuously monitor an executing Ada program for specification consistency [22]. The user manually annotates an Ada program with constructs from ANNA, a formal specification language. Mok and Liu [17] proposed an approach for monitoring the

violation of timing constraints written in the specification language based on Real-time Logic as early as possible with low-overhead. The framework we describe in this paper does not limit itself to any particular kind of monitored properties. In [15], an elaborate language for specification of monitored events based on relational algebra is proposed. Instrumentation of high-level source code is provided automatically. Collected data are stored in a database. Since the instrumentation code performs database queries, instrumentation can significantly alter the performance of a program.

A large body of related research work concentrates on automated generation of *test oracles* from the requirements. A general methodology for doing this is discussed in [21], together with examples in Real Time Interval Logic (RTIL) and Z. In [2] a trace analysis tool for LOTOS requirements is described, while [7] describes a similar tool for Estelle requirements. Generating test oracles for Graphical Interval Logic (GIL) is discussed in [6, 18]. An equivalent problem for a safe fragment of Linear Temporal Logic is put forth in [11]. This fragment is expressively similar to MEDL. We note that the MaCS framework gives more than just a test oracle for a given specification. Its ability to generate diagnostic information and provide feedback the the system in case of requirement violations makes it a more general tool.

The simulation and monitoring platform MT-Sim [3], based on the graphical real-time specification language Modechart, is similar in its intent to MaCS, however, we are not as tied to a fixed specification formalism.

6 Conclusions

This paper describes the Monitoring-aided Checking and Steering (MaCS) framework which is developed to assure the correctness of execution at run-time and to perform dynamic correction of system behavior by steering actions. Monitoring and checking is performed based on a formal specification of system requirements, and is use to detect violations of safety properties in the observed execution of the monitored system. Steering is integrated with monitoring and checking to put the system back to a safe state. The MaCS framework is a step towards bridging the gap between verification of system design specifications and validation of system implementations in a high-level programming language. The former is desirable but yet impractical for large systems, while the latter is necessary but informal or incomplete.

There are several issues that need further work. For example, we would like to understand better the theoretical basis for steering; in particular, what problems can be resolved by means of steering, what is the right way to reason about steering, to convince ourselves that steering will have the desired effect. The current prototype system for Java is available at `www.cis.upenn.edu/~rtg/macs`, and will serve as an important vehicle in exploring the possibilities and shortcomings of our approach. We are currently conducting several case studies in monitoring and steering of systems, and we expect to gain much experience from them. We also plan to extend the prototype implementation to support distributed systems written in Java.

References

[1] P.C. Bates and J.C. Wileden. High-level debugging: The behavioral abstraction approach. *J. Syst. Software*, 3(255-264), 1983.

[2] G.v. Bochmann, R. Dssouli, and J.R. Zhao. Trace analysis for conformance and arbitration testing. *IEEE Transactions on Software Engineering*, 15(11):1347–1356, November 1989.

[3] Monica Brockmeyer, Farnam Jahanian, Constance Heitmeyer, and Bruce Labaw. A flexible, extensible environment for testing real-time specifications. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1997.

[4] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[5] Michel Diaz, Guy Juanole, and Jean-Pierre Courtiat. Observer - a concept for formal on-line validation of distributed systems. *IEEE Transactions on Software Engineering*, 20(12):900–913, December 1994.

[6] Laura K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'94)*, volume 19, pages 140–153, December 1994. Proceedings published as Software Engineering Notes.

[7] S.A. Ezust and G.v. Bochmann. An automatic trace analysis tool generator for estelle specifications. *Computer Communication Review*, 25(4):175–184, October 1995. Proceedings of ACM SIGCOMM 95 Conference.

[8] Diana Gordon, William Spears, Oleg Sokolsky, and Insup 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*, November 1999.

[9] C. Heitmeyer and D. Mandrioli, *Eds. Formal Methods for Real-Time Systems.* Number 5 in Trends in Software. John Wiley & Sons, 1996.

[10] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. Scr*: A toolset for specifying and analyzing requirements. In *Proceedings of COMPASS*, 1995.

[11] L. J. Jagadeesan, A. Porter, C. Puchol, J. C. Ramming, and L.G.Votta. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the International Conference on Software Engineering*, May 1997.

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

[13] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. A framework for run-time correctness assurance of real-time systems. Technical Report MS-CIS-98-37, University of Pennsylvania, 1998.

[14] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, 1999.

[15] Yingsha Liao and Donald Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Transactions on Software Engineering*, 18(11):969–979, November 1992.

[16] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[17] Aloysius K. Mok and Guangtian Liu. Efficient run-time monitoring of timing constraints. In *IEEE Real-Time Technology and Applications Symposium*, June 1997.

[18] T.O. O'Malley, D.J. Richardson, and L.K. Dillon. Efficient specification-based test oracles. In *Second California Software Symposium (CSS'96)*, April 1996.

[19] J. Peleska. Test automation for safety-critical systems: Industrial application and future developments. In *FME'96: Third International Symposium of Formal Methods Europe*, volume 1051 of *LNCS*, pages 39–59, 1996.

[20] D. K. Peters and D. L. Parnas. Requirements-based monitors for real-time systems. In *ISSTA'00: International Symposium on Software Testing and Analysis*, 2000.

[21] D.J. Richardson, S. Leif Aha, and T.O. O'Malley. Specification-based oracles for reactive systems. In *14th International Conference on Software Engineering*, May 1992.

[22] Sriram Sankar and Manas Mandal. Concurrent runtime monitoring of formally specified programs. In *IEEE Computer*, pages 32 –41, March 1993.

[23] T. Savor and R. E. Seviora. An approach to automatic detection of software failures in real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 136 –146, June 1997.

[24] Beth A. Schroeder. On-line monitoring: A tutorial. In *IEEE Computer*, pages 72 – 78, June 1995.