Runtime Assurance Based On Formal Specifications*

I. Lee†, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

March 23, 1999

Abstract

We describe the Monitoring and Checking (MaC) framework which assures the correctness of the current execution at run-time. Monitoring is performed based on a formal specification of system requirements. MaC bridges the gap between formal specification and verification, which ensures the correctness of a design rather than an implementation, and testing, which partially validates an implementation. An important aspect of the framework is a clear separation between implementation-dependent description of monitored objects and high-level requirements specification. Another salient feature is automatic instrumentation of executable code. The paper presents an overview of the framework and two languages to specify monitoring scripts and requirements, and briefly explain our on-going prototype implementation.

1 Introduction

Much research in the past two decades concentrated on methods for analysis and validation of distributed and real-time systems. 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 prevalent 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 formal design. So, there are possibilities for introduction of errors into an implementation of the design that has been verified. One way that people have traditionally tried to overcome this gap between design and implementation has been to test an implementation on a pre-determined set of input sequences. This approach, however, fails to provide guarantees about the correctness of the implementation on all possible input sequences. Consequently, when a system is running, it

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

†Corresponding Author. Insup Lee, email: lee@cis.upenn.edu; fax: +1(215) 573-3573

is hard to guarantee whether or not the current execution of the system is correct using the two traditional methods. Therefore, the approach of continuously monitoring a running system has can be used to fill the gap between these two approaches.

In this paper, we describe a framework of monitoring and checking a running system with the aim of ensuring that it is running correctly with respect to a formal requirements specification. The use of formal methods is the salient aspect of our approach. We concentrate on the following two issues: (1) how to map high-level abstract events that are used in requirement specification to low-level activities of a running system, and (2) how to instrument the code to extract and detect necessary low-level activities. We assume that both requirement specifications and the system implementation are available to us.

The major phases of the framework are as follows: (1) system requirements are formalized; at the same time, a *monitoring script* is constructed, which is used to instrument the code and establish a mapping from low-level information into high-level events; (2) at run-time, events generated by the instrumented system are monitored for compliance with the requirements specification. The run-time monitoring and checking (MaC) architecture consists of three components: *filter, event recognizer*, and *run-time checker*. The filter extracts low-level information (such as values of program variables and time when variables change their values) from the instrumented code. The filter sends this information to the event recognizer, which converts it into high-level events and conditions and passes them to the run-time checker.

Each event delivered to the checker has a timestamp, which reflects the actual time of the occurrence of the event. This enables us to monitor real-time properties of the system. Timestamps are assigned to events by the event recognizer based on the clock readings provided by the filter. The run-time checker checks the correctness of the system execution *thus far* according to a requirements specification of the system, based on the information it receives from the event recognizer, and on the past history. The checker can combine monitoring of behavioral correctness of the system control flow with program checking [2] for numerical computations. This integrated approach is a unique feature of the proposed framework. The current prototype implementation of the MaC framework supports the monitoring of a system written in Java. Instrumentation is performed automatically, directly in Java bytecode.

Related work. Computer systems are often monitored for performance measurement, evaluation and enhancement as well as to help debugging and testing [17]. 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. 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. The authors of [16] also consider only input/output behavior of the system. In our opinion, 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 [14] is also targeted towards monitoring of black-box systems without resorting to instrumentation. Additionally, we aim at using the MaC framework beyond testing, during real system executions. Sankar and Mandel have developed a methodology to continuously monitor an executing Ada program for specification consistency [15]. The user manually annotates an Ada program with constructs from ANNA, a formal specification language. Mok and Liu [12] 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 proposed in this paper does not limit itself to any particular kind of monitored properties. In [10], an elaborate language for specification of monitored events based on relational algebra is proposed. Similarly to our approach, the authors try to minimize effects of instrumentation on run-time performance, and to reduce the instrumentation cost through automated instrumentation. Their goal, however, goes beyond run-time monitoring. For our purposes, a simpler and easier to interpret event description language of MaC appears to be more appropriate.

The paper is organized as follows. Section 2 presents an overview of the framework. Section 3 informally presents the language for monitoring scripts and requirements specifications. Section 4 briefly overviews a prototype implementation of the MaC framework as well as the current future plans. More complete and formal treatment of the MaC framework is given in [9].

2 Overview of the MaC Framework

The MaC framework aims at run-time assurance monitoring of real-time systems. The structure of the framework is shown in Figure 1. The framework includes two main phases: (1) before the system is run, its implementation and requirement specification are used to generate run-time monitoring components; (2) during system execution, information about the running system is collected and matched against the requirements.

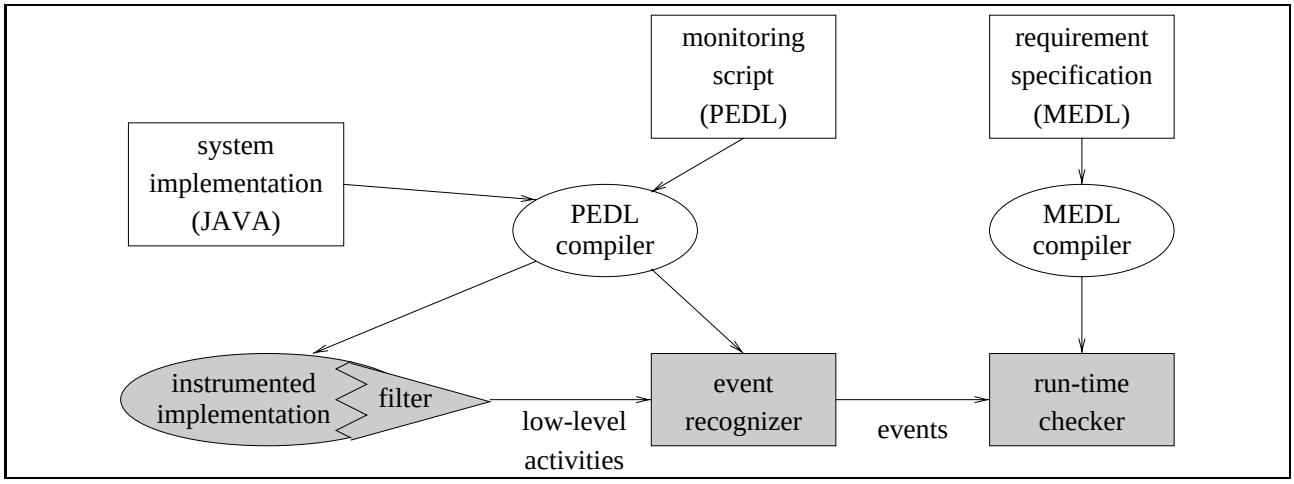

Figure 1: Overview of the MaC framework

A major task during the first phase (indicated by clear boxes in Figure 1) is to provide a mapping between high-level events used in the requirement specification and low-level state information extracted during execution. They are related explicitly by means of a *monitoring script*. The monitoring script describes how events at the requirements level are defined in terms of monitored states of an implementation. For example, in a gate controller of a railroad crossing system, the requirements may be expressed in terms of the event `train_in_crossing`. The implementation, on the other hand, stores the train's position with respect to the crossing in a variable `train_position`. The monitoring script in this case can define the event as condition `train_position < 800`. The language of monitoring script (described in Section 3) has limited expressive power in order to ensure fast recognition of events.

The monitoring script is used to generate a *filter* and an *event recognizer* automatically. The filter instruments the implementation to extract the necessary state information at run-time. The event recognizer receives state information from the filter and determines the occurrences of event according to the event definition in the script. Also, a *run-time checker* is generated from the formal requirements. The requirement specification uses events defined in the monitoring script.

During the run-time phase (shaded boxes in Figure 1), the instrumented implementation is executed while being monitored and checked against the requirements specification. The filter sends relevant state information to the event recognizer, which determines the occurrence of events. These events are then relayed to the run-time checker to check adherence to the requirements.

Filter. A filter is a set of program fragments that are inserted into the implementation to instrument the system. The essential functionality of a filter is to keep track of changes to monitored objects and send pertinent state information to the event recognizer. Instrumentation is performed directly on the executable code (bytecode, in the case of Java). Instrumentation is automatic, which is made possible by the low-level description in the monitoring script.

Event Recognizer. The event recognizer is the part of the monitor that detects an event from values of monitored variables received from the filter. Events are cognized according to a monitoring script (written in PEDL) and recognized events are sent to the run-time checker. Each event is supplied with a timestamp that can be used in checking real-time properties. Events may additionally have associated numerical values to facilitate program checking by the monitor. 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 framework.

Run-time Checker. The run-time checker determines whether or not the current execution history satisfies the given requirements (written MEDL). The execution history is captured from a sequence of events sent by the event recognizer. The checker can handle behavioral as well as numerical requirements. The latter can be analyzed using the technique of program checking. It may seem that the detection of a requirement violation at run-time is too late for recovery. This, however, is not necessarily true. A monitored property may represent a potentially dangerous condition that needs an attention from a human operator, which is the function that the run-time checker provides.

3 The MaC Language

In this section, we give a brief overview of the languages used to describe what to observe in the program and the requirements the program must satisfy. The scripts written in these languages are then used to automatically generate the event recognizer and the run-time checker, respectively.

The language for monitoring scripts is called PEDL (**P**rimitive **E**vent **D**efinition **L**anguage, Section 3.4). PEDL scripts are used to define what information is sent from the filter to the event recognizer, and how they are transformed into requirements-level events by the event recognizer. Requirement specifications are written in MEDL (**M**eta **E**vent **D**efinition **L**anguage, Section 3.5). The primary reason for having two separate languages in the monitoring framework is to separate implementation-specific details of monitoring from requirements specification. This separation ensures that the framework is scalable to different implementation languages and specification

formalisms, while providing a clean interface to the designer of monitors. For example, if we wish to retarget our system from programs written in Java to C++, then all we would need to modify is the syntax of PEDL, leaving MEDL unchanged.

Before presenting the two languages, PEDL and MEDL, we discuss some key issues in the logical semantics of these languages. In Section 3.1, we illustrate the distinction between *events* and *conditions*. In Section 3.2, we discuss how the language may handle the presence of variables that are not defined. We then formalize our intuitions into a logic in Section 3.3. This logic provides the formal foundations for PEDL (in Section 3.4) and MEDL (in Section 3.5).

3.1 Events and Conditions

As described in Section 2, whenever an "interesting" state change occurs in the running system, the filter sends a notification to the monitor. Based on updates from the filter, the monitor matches the trace of the current execution against the requirements. In order to do this, we distinguish between two kinds of state information underlying the notifications.

Events occur instantaneously during the system execution, whereas *conditions* are information that hold for a duration of time. For example, an event denoting return from method `RaiseGate` occurs at the instant the control returns from the method, while a condition `(position == 2)` holds as long as the variable position does not change its value from `2`. 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. For an event, 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.

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. Conditions, on the other hand, have *durations*, intervals 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 later when we present the logic.

3.2 Presence of Undefined Variables

Reconsider the condition `(position == 2)` that was used previously. When the variable `position` has some integer value, it is very clear what this condition means. However, before the variable `position` is initialized at the start of the execution, it is not clear whether this condition should be considered to be true or false. This problem is not just confined to the start. During any execution, variables routinely become undefined when they are out of scope, and if we want to reason about such variables then we need a consistent way of intepreting logical formulae having undefined variables. The problems associated with defining the semantics of logics in the presence of partial functions[1] are well-understood [6, 3, 13]. There have been some approaches to defining logics with partial functions where the formulae are interpreted over boolean values, i.e., true and false. However, these approaches work only when the logic has no primitive relations, like "$<$" and "$\geq$", which have some "natural" interpretation. Another traditional approach towards handling undefined expressions, has been to move to a three-valued logic, where the third value is taken to

[1] Variables can be thought of as partial functions over time

represent undefined. We choose to take this later approach, and so interpret the truth of conditions over a three-valued logic.

We now formalize the issues presented above, in a two-sorted logic that defines the operations on events and conditions. In this logic, we shall interpret conditions over three values and not over booleans. PEDL and MEDL are subsets of this logic with added means of definition of primitive events and conditions.

3.3 Logic for Events & Conditions

Syntax. We assume a countable set $\mathcal{C} = \{c_1, c_2, \ldots\}$ of primitive conditions. For example, in the monitoring script language (Section 3.4), these primitive conditions will be Java boolean expressions built from the values of the monitored variables. In the requirements description language (Section 3.5) these will be conditions that were recognized by the event recognizer and sent to the run-time checker.

We also assume a countable set $\mathcal{E} = \{e_1, e_2, \ldots\}$ of primitive events. When an event occurs (to be defined formally later), it can have an attribute value, which is an element of a set $\mathcal{S}_{e_i}$. For example, `StartM(RaiseGate)` is a primitive event in the monitoring script language, which is present at the start of method `RaiseGate` and whose attribute value is the tuple of values of all the parameters with which this method is called. The primitive events in the requirements description language are those that are reported by the event recognizer.

The logic has two sorts: conditions and events. The syntax of conditions (C) and events (E) is as follows:

$\langle C \rangle ::= c \mid \text{defined}(\ \langle C \rangle\ ) \mid [\ \langle E \rangle\ ,\ \langle E \rangle\ ) \mid !\ \langle C \rangle \mid \langle C \rangle\ \&\&\ \langle C \rangle \mid \langle C \rangle\ ||\ \langle C \rangle \mid \langle C \rangle \Rightarrow \langle C \rangle$

$\langle E \rangle ::= e \mid \text{start}(\ \langle C \rangle\ ) \mid \text{end}(\ \langle C \rangle\ ) \mid \langle E \rangle\ \&\&\ \langle E \rangle \mid \langle E \rangle\ ||\ \langle E \rangle \mid \langle E \rangle\ \text{when}\ \langle C \rangle$

Semantics. The models for this logic are sequences of worlds, similar to those used for linear temporal logic. Each world has a description of the truth values of primitive conditions and occurrences of promitive events. More formally, a model M is a tuple (S, τ, L_C, L_E), where $S = \{s_0, s_1, \ldots\}$, τ is a mapping from S to the time domain (which could be integers, rationals, or reals), L_C is a total function from $S \times \mathcal{C}$ to $\{\mathbf{true}, \mathbf{false}, \Lambda\}$, and L_E is a partial function from $S \times \mathcal{E}$ to $\mathcal{D}_e$. Intuitively, L_C assigns to each state the truth values of all the primitive conditions; since we shall interpret conditions over a 3-valued logic, the truth value of primitive conditions can be **true**, **false** or Λ (undefined). Similarly, in each state s, $L_E(s, e)$ is defined for each event e that occurs at s and gives the value of the primitive event e. The mapping τ defines the time at each state, and it satisfies the requirement that $\tau(s_i) < \tau(s_j)$ for all $i < j$, i.e., the time at a later state is greater.

In order to define what we mean by a condition c being true in model M at time t ($M, t \models c$), we need to define what we mean by its denotation ($\mathcal{D}^t_M(c)$). This is defined in Figure 3.3. Using this we define the meaning of $M, t \models c$, and of an event e occurring in a model M at time t ($M, t \models e$). The formal definition is given in Figure 3.3. [2]

As stated before, we shall interpret conditions over three values, **true**, **false**, and Λ (undefined). The denotation of a primitive condition, c at time t is given by c's truth value in the last state before time t. The predicate defined(c) is true whenever the condition c has a well-defined value, namely, **true** or **false**. The denotation of negation ($!c$), disjunction ($c_1||c_2$) and conjunction ($c_1\&\&c_2$) are

[2] Notice, that the definition of $\mathcal{D}^t_M$ refers to the definition of $\models$, and vice versa. However, the definitions are well-defined.

$$
\begin{array}{ll}
[c_k \text{ primitive}] & \mathcal{D}^t_M(c_k) = L_C(s_i, c_k), \text{ where } \tau(s_i) \leq t \text{ and for all } s_j \ (j > i)\ \tau(s_j) > t \\
[\text{defined}] & \mathcal{D}^t_M(\text{defined}(c)) = \begin{cases} \textbf{true} & \text{if } \mathcal{D}^t_M(c) \neq \Lambda \\ \textbf{false} & \text{otherwise} \end{cases} \\
[\text{pair}] & \mathcal{D}^t_M([e_1, e_2)) = \begin{cases} \textbf{true} & \text{if there exists } t_0 \leq t \text{ such that } M, t_0 \models e_1 \text{ and for all} \\ & \quad t_0 \leq t' \leq t,\ M, t' \not\models e_2 \\ \textbf{false} & \text{otherwise} \end{cases} \\
[\text{negation}] & \mathcal{D}^t_M(!c) = \begin{cases} \textbf{true} & \text{if } \mathcal{D}^t_M(c) = \textbf{false} \\ \Lambda & \text{if } \mathcal{D}^t_M(c) = \Lambda \\ \textbf{false} & \text{if } \mathcal{D}^t_M(c) = \textbf{true} \end{cases} \\
[\text{disjunction}] & \mathcal{D}^t_M(c_1||c_2) = \begin{cases} \textbf{true} & \text{if } \mathcal{D}^t_M(c_1) \text{ or } \mathcal{D}^t_M(c_2) \text{ is } \textbf{true} \\ \textbf{false} & \text{if } \mathcal{D}^t_M(c_1) = \mathcal{D}^t_M(c_2) = \textbf{false} \\ \Lambda & \text{otherwise} \end{cases} \\
[\text{conjunction}] & \mathcal{D}^t_M(c_1 \&\& c_2) = \mathcal{D}^t_M(!(!c_1||!c_2)) \\
[\text{implication}] & \mathcal{D}^t_M(c_1 \Rightarrow c_2) = \mathcal{D}^t_M(!c_1||c_2)
\end{array}
$$

Figure 2: Denotation for conditions

$M, t \models c$	iff	$\mathcal{D}^t_M(c) = \textbf{true}$
$M, t \models e_k$ (e_k primitive)	iff	there exists state s_i such that $\tau(s_i) = t$ and $L_E(s_i, e_k)$ is defined.
$M, t \models \text{start}(c)$	iff	$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i) \models c$ and $M, \tau(s_{i-1}) \not\models c$. i.e., start($c$) occurs when condition c changes from false to true.
$M, t \models \text{end}(c)$	iff	$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i) \not\models c$ and $M, \tau(s_{i-1}) \models c$. i.e., end($c$) occurs when condition c changes from true to false.
$M, t \models e_1 \| e_2$	iff	$M, t \models e_1$ or $M, t \models e_2$.
$M, t \models e_1$ && e_2	iff	$M, t \models e_1$ and $M, t \models e_2$.
$M, t \models e$ when c	iff	$M, t \models e$ and $M, t \models c$. i.e., event e occurs when condition c is true.

Figure 3: Semantics of events and conditions.

interpreted classically whenever c, c_1 and c_2 take values **true** or **false**; the only non-standard cases are when these take the value Λ. In these cases, we interpret them as follows. Negation of an undefined condition is Λ. Conjunction of an undefined condition with **false** is **false**, and with **true** is Λ. Disjunction is defined dually; disjunction of undefined condition and **true** is **true**, while disjunction of undefined condition and **false** is Λ. Implication ($c_1 \Rightarrow c_2$) is taken to $!c_1||c_2$.

For primitive events, once again, the truth value is given by the labels on the states. Conjunction ($e_1 \&\& e_2$) and disjunction ($e_1||e_2$) defined classically; so $e_1 \&\& e_2$ is present only when both e_1 and e_2 are present, whereas $e_1||e_2$ is present when either e_1 or e_2 is present.

There are some natural events associated with conditions, namely, the instant when the condition becomes **true** (start(c)), and the instant when the condition becomes **false** (end(c)). Notice, that the event corresponding to the instant when the condition becomes Λ can be described as end(defined(c)). Also, any pair of events define an interval of time, so forms a condition $[e_1, e_2)$ that is **true** from event e_1 *until* event e_2. Finally, the event (e when c) is present if e occurs at a time when condition c is **true**.

Notice that every condition can be identified with the events corresponding to when it becomes **true**, when it becomes **false** and when it becomes Λ. This is the reason why the languages in the MaC framework, are called "event definition languages".

3.4 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.

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 Section 3.6.

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

```
Cond TooFast = Train.calculatePosition().trainSpeed > 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*.

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 3.3, 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. If the monitored system has several clocks, we assume, for this paper, that the clocks are perfectly synchronized to simplify the presentation of this paper.

3.5 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, described in Section 3.3. 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 is Section 3.3 has a 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 3.3. These events and conditions are then used to define safety properties and alarms.

```
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

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 [11] 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.

3.6 Example

We illustrate the use of PEDL and 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 [7]. 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.

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 Entities:
    void GateController.open();
```

```
    void GateController.close();
    int  GateController.gatePosition;
CondDef:
    Cond Gate_Down = (GateController.gatePosition == GateController.GATE_DOWN);
EventDef:
    Event OpenGate =  StartM( GateController.open() );
    Event CloseGate = StartM( GateController.close() );
```

The correctness requirement for the gate is given in the MEDL script below. 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.

```
import event OpenGate, CloseGate;
import condition Gate_Down;
AuxVarDecl:
    float lastClose;
    float currentTime;
SafePropDef:
    Cond GateClosing =
         [ CloseGate when !Gate_Down, OpenGate || start(Gate_Down) )
         => lastClose + 30 > currentTime;
AuxVarDefL
    CloseGate -> lastClose := time(CloseGate);
```

4 Conclusions

This paper describes the Monitoring and Checking (MaC) framework which is developed to assure the correctness of an execution at run-time. Monitoring is performed based on a formal specification of system requirements. The MaC 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 and error-prone.

We are currently implementing a prototype of the MaC framework for systems written in Java. Given a PEDL script and an implementation in Java bytecode, the PEDL compiler inserts filters into the bytecode for instrumentation. The PEDL compiler also generates event definitions which are interpreted by the event recognizer. The MEDL compiler generates a run-time checker from a MEDL specification. In addition to working on this prototype system, we are exploring how to translate requirements written in a logic such as RTL [8] or Linear Logic to MEDL. Finally, we are also investigating how to extend the MaC framework to support the *steering* of a monitored system to a safe state.

References

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

[2] M. Blum and S. Kannan. Designing programs that check their work. In *JACM V.42 No. 1*, pages 269 –291, January 1995.

[3] J. Cheng and C. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. Woodstock, editors, *Proceedings of Third Refinement Workshop*. Springer-Verlag, 1991.

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

[5] M. Diaz, G. Juanole, and J.-P. Courtiat. Observer - a concept for formal on-line validation of distributed systems. *IEEE Transactions on Software Engineering*, 20(12):900–913, Dec. 1994.

[6] W. F. Farmer. A partial functions version of church's simple theory of types. *Journal of Symbolic Logic*, pages 1269 – 1291, September 1990.

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

[8] F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[9] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. A framework for run-time correctness assurance of real-time systems. Technical Report MS-CIS-98-37, University of Pennsylvania, 1998.

[10] Y. Liao and D. Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Transactions on Software Engineering*, 18(11):969–979, Nov. 1992.

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

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

[13] D. L. Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19(9):856 – 861, September 1993.

[14] 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.

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

[16] 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.

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