Formally Specified Monitoring of Temporal Properties*

Moonjoo Kim, Mahesh Viswanathan,
Hanêne Ben-Abdallah,[†] Sampath Kannan, Insup Lee, and Oleg Sokolsky[‡]
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Abstract

We describe the Monitoring and Checking (MaC) framework which provides assurance on the correctness of an execution of a real-time system at runtime. Monitoring is performed based on a formal specification of system requirements. MaC bridges the gap between formal specification, which analyzes designs rather than implementations, and testing, which validates implementations but lacks formality. 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, languages to express monitoring scripts and requirements, and a prototype implementation of MaC targeted at systems implemented in Java.

1 Introduction

Real-time systems often arise in the area of embedded and safety-critical applications. Dependability of such systems is the utmost concern to their developers. Much research in the past two decades concentrated on methods for analysis and validation of 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.

Still, complete formal verification has not yet become a prevalent method of analysis. The reasons for this are twofold. First, full 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 specifications of these systems. Construction of such specifications is usually a manual and error-prone process. Separate methods are needed, then, to verify compliance of the system implementation to its formal specification. Testing, on the other hand, allows one to validate the system implementation directly. However, testing results lack the rigor of formal analysis and usually do not provide guarantees of absence of errors in the implementation.

Consequently, whichever analysis approach has been taken to validate a real-time system, there exists a possibility of incorrect behavior during the execution of the system. Run-time monitoring and checking strives to address this problem.

Computer systems are often monitored for performance evaluation and enhancement [10], debugging and testing [14], and to control or check of system correctness [18]. Recently, the problem of designing monitors to check for the correctness of system implementation has received increased attention from the research community [3, 15, 16, 13, 17]. Such monitors can be used to detect violations of timing [13] or logical [3] properties of a program, constraints on language constructs [15], and so on.

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 architecture, monitors systems written in Java. Instrumentation is performed automatically, directly in JAVA bytecode. A language called MEDL, based on a linear temporal logic, is used to describe the formal requirements. Other formal languages can be readily used to specify requirements.

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. 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 [13] 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. 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.

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 describes a prototype implementation of the MaC framework. More complete and formal treatment of MaC 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.

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

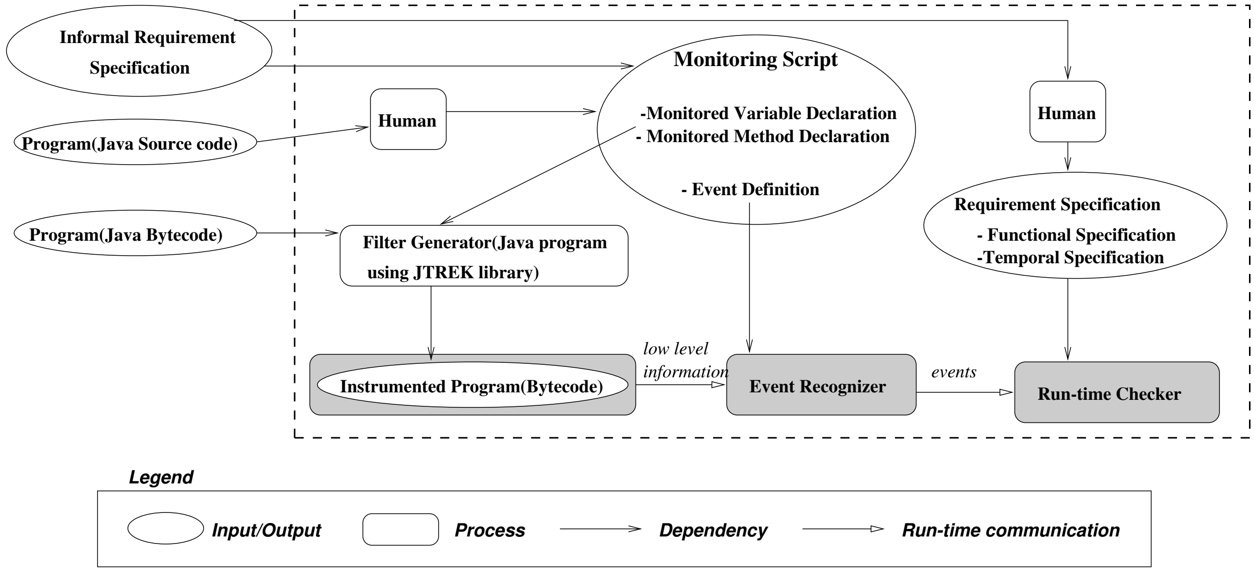

Figure 1. Overview of the MaC framework

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 scripts event recognizer (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 events according to their definition in the script. Also during the first phase, the system requirements are formalized, and a *run-time checker* is produced 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 statically 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 according to the monitoring script. Recognized events are delivered 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.

While it is conceivable to merge the event recognizer with the filter, we chose to separate the two modules. The separation allows us to remove the overhead of abstracting out events from the low-level information. This reduces interference of the monitor with the monitored system's execution. On the other hand, communication overhead incurred by sending changes in the monitored data from the filter to the event recognizer increases, but it applies only to the off-line processing of the monitored information and is therefore more acceptable. An additional advantage of the chosen design is a clear separation of monitoring activity from the system activity.

Run-time checker. The run-time checker checks that the current execution satisfies the given requirements, based on the information provided by the event recognizer. The checker can handle behavioral as well as numerical requirements. The latter are analyzed using the technique of program checking. The prototype implementation does not have provisions for program checking yet. The current implementation uses language MEDL (see Section 3.4) to express requirements.

It may seem that a violation of a requirement at run time is a catastrophic event, and that it is too late to recover from it. This, however, is not necessarily true. A monitored property may represent a potentially dangerous condition that need to draw the attention of a human operator, which is the function that the run-time checker provides. We illustrate this concept with an example in Section 3.5.

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 3.3). 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 3.4). 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.

Objects described in both PEDL and MEDL scripts are *events* and *conditions*. Before we present the two languages, we illustrate the distinction between events and conditions.

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 the 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. For example, an event denoting return from method `RaiseGate` occurs at the instant the control returns from the method. We can conclude that this event does not occur at any moment except when the monitor receives an update from the filter. By contrast, *conditions* may hold between updates. Consider monitoring condition `(position == 2)`. Once the monitor receives a message from the filter that variable `position` has been assigned the value `2`, we can conclude that it keeps this value until the next update comes.

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 below.

This distinction between events and conditions is formalized in a simple two-sorted logic that defines various operations on events and conditions. PEDL and MEDL are subsets of this logic with added means of definition of primitive events and conditions.

3.2 A 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 PEDL, these primitive conditions will be Java boolean expressions built from the values of the monitored variables. In the requirements description language MEDL these will be conditions that were recognized by the event recognizer and sent to the checker.

We also assume a countable set $\mathcal{E} = \{e_1, e_2, \ldots\}$ of primitive events. When an event occurs, it can have an attribute value, which is an element of a set $\mathcal{D}_{e_i}$. For example, `startM (RaiseGate)` is a primitive event in the monitoring script language, which occurs 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 will be 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 | [$\langle E \rangle$, $\langle E \rangle$) | ! $\langle C \rangle$ | $\langle C \rangle$ && $\langle C \rangle$ | $\langle C \rangle$ || $\langle C \rangle$ | $\langle C \rangle \Rightarrow \langle C \rangle$

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

Semantics. The models for this logic are similar to those for linear temporal logic, in that they are a sequence 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 filter (or event recognizer) sends updates, and so these models are a discrete abstraction of the execution of the running system.

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 ($c_1 \&\& c_2$), disjunction ($c_1 || c_2$) and implication ($c_1 \Rightarrow c_2$) of conditions is defined naturally; so $!c$ is true when c is false, $c_1 \&\& c_2$ is true only when both c_1 and c_2 are true, $c_1 || c_2$ is true when either c_1 or c_2 is true, and $c_1 \Rightarrow c_2$ is true if c_2 is true whenever c_1 is true. Conjunction ($e_1 \&\& e_2$) and disjunction ($e_1 || e_2$) 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 $[e_1, e_2)$ that is true from event e_1 until e_2. Finally, the event e when c is true if e occurs and condition c is true at that time instant.

The formal semantics for this logic is given in [9].

Notice that some natural equivalences hold in this logic. For example, for any condition c, $c \equiv [start(c), end(c))$. This allows one to identify conditions with pairs of events, and is the reason why the languages in the MaC framework, are called "event definition languages". Also, for conditions c_1 and c_2, and event e, e when c_1 when $c_2 \equiv e$ when (c_1 && c_2).

3.3 Primitive Event Definition Language (PEDL)

PEDL is the language for writing monitoring scripts. 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 of the language reflect 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, 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 4.

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 method `f` (resp., returns from `f`. 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 3.2, 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.

3.4 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 3.2. 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 3.2 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. Precisely, 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 3.2. These events and conditions are then used to define the 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 [12] 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.

3.5 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

```
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 2. Implementation of the gate controller

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 2 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 `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 The Current MaC Prototype System

This section introduces a prototype implementation of the MAC framework. The prototype closely follows the general architecture (see Figure 1 in Section 2). We discuss implementation aspects of the filter, the event recognizer and the run-time checker.

4.1 Filter and Code Instrumentation

Java bytecode has been selected as the basis for instrumentation for the following reasons: (1) a class file, the unit of Java bytecode, contains rich symbolic information about the system [11] that can be used for automatic instrumentation; (2) Java bytecode is strongly typed and excludes pointer arithmetic; (3) growing popularity of Java, combined with platform independence of Java bytecode will make the framework widely applicable. In addition, there are many high languages like Ada and Lisp which compile its source code into Java bytecode [19].

Several aspects of the presented framework present implementation challenges. We now briefly outline these challenges and describe the limitations of the prototype.

Naming of monitored objects. In order to specify monitored entities unambiguously, we use hierarchical names constructed from identifiers used in the source code. The following meaning is ascribed to a name `x.y`: (1) if `x` is a class name, `y` is a field or a method of the class. If `y` is not static, it will apply to every instance of `x`; (2) if `x` is a variable of type `T`, `y` is a field or a method of class `T`; (3) if `x` is a method of class `T`, `y` is a local variable of `x`. Unless `y` is of a primitive (non-reference) Java type, the name can be extended further according to the same rules. Examples of declarations include `RRC.train_x`, `Train.position().trainSpeed`, and `Gate.gateDown()`. The first of these declarations denotes field `train_x` of class `RRC`, the second identifies local variable `trainSpeed` of method `position` in class `Train`. The last one identifies a method in class `Gate`.

Detection of object updates. We need to guarantee that all updates to a monitored object are reported. Two problems need to be addressed: *aliasing*, where an object can be monitored through several references, and *reference changing*, where a reference is modified to refer to an object that was not intended to be monitored.

In general, we do not know statically which references refer to the object of interest. We have, therefore, to check accesses through all reference variables of the same type. The user has the option to enable this feature explicitly. However, most of the monitored objects in the examples that we have considered are always accessed through the same reference. Therefore, we chose to disable the feature by default.

A filter consists of a set of code fragments inserted into class files of the target system and Java class, which provides for storing update information and communication between filter and event recognizer. The following kinds of program entities can be instrumented:

1) Execution points. The current prototype detects when the execution point reaches a method invocation, return from the method, start and end of program and exception of method. Invocation and return from the method can be detected by inserting the instrumentation before the first instruction and after the last instruction of the code for the method. Exceptions are monitored by instrumenting the exception table of the method.

2) Local and field variables. Every local variable is always accessed through an index *fixed* in a bytecode instruction. There are only two kinds of instructions that may modify a local variable. Instrumentation is inserted immediately following the update. Similarly, there is only one instruction that accesses a field variable in an object, and the access is through a fixed parameter in the bytecode.

The filter is generated by *filter generator* which is written in Java using JTrek library [6] for inserting bytecode fragments into the program. The filter generator gets a program which is to be instrumented and a list of monitored variables and monitored methods as input. It generates the instrumented program as output by inserting codes at proper places of the program. The filter sends updated values, together with a timestamp and identification of the thread that occasioned the update, whenever it detects an updating of a monitored entity. To minimize the overhead to the system, sending values to event recognizer through the network is performed by a separate thread.

4.2 Event recognizer

The event recognizer translates low-level state changes communicated by the filter, into high-level events and conditions. The event recognizer maintains a table that stores the current value for each monitored variable. Each message from the filter causes this table to be updated.

Whenever an update from the filter arrives, the event recognizer re-evaluates the truth of all events and conditions. Conditions defined in terms boolean expressions over the monitored variables can be directly evaluated from the table of current values of all monitored variables. However, in order to identify events `start(c)` and `end(c)`, it must not only know the current truth value of condition `c`, but also its truth value at the time of the previous update. The same is true for the event `end(c)`. Hence the checker also keeps track of the values of all the conditions at the time of the previous update, in addition to the values of the monitored objects. Finally, once the recognizer has determined the truth of all the conditions and events defined in the monitoring script, it sends to the checker its "exported" events and changes in "exported" conditions.

4.3 Run-time Checker

The checker maintains a timed trace of the current execution based on the messages received from the event recognizer. Each event is supplied with a timestamp, reflecting the time when the event occurred. Each value of the timestamp introduces a new state in the time trace. At each state, event occurrences and the values of conditions are evaluated. As several received events may have the same timestamp, evaluation of a state is deferred until all events with the same timestamp arrive. The checker is guaranteed to receive messages with non-decreasing timestamps. In the prototype, TCP/IP protocol is used for communication between the event recognizer and the checker to ensure proper sequencing. The truth value of every event and condition can be evaluated in constant time in terms of the length of the trace and linear in the size of the requirement specification.

Once all the truth of all the events and conditions has been determined, the auxiliary variables are updated. If the event guarding the update occurs at the current state, the auxiliary variable is updated as per the assignment rule. Whenever an alarm becomes true or a safety property becomes false, the checker declares the program to be incorrect.

5 Conclusions

The paper makes 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 efficient but informal and error-prone.

To this end, we have presented a design and a prototype implementation of an on-line monitoring of correctness properties of real-time systems. Monitoring is based on formally specified system requirements. The formality of approach guarantees that at least the current execution complies with the requirements. A variety of specification formalisms can be easily accommodated in the framework. For example, all properties expressed in real-time logic RTL [8] can be efficiently checked.

The immediate goals of the future work on this topic include extensions of the prototype into a full-strength monitoring system and extension of the framework to other languages beyond Java. Another avenue of research is aimed at a transition from passive observation to active guidance of the monitored system. Our current system is geared towards the *detection* of faults. It would be desirable in future to build monitors that can *steer* a system to a correct 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] S. E. Chodrow and M. G. Gouda. The Sentry System. In *SRDS11*, October 1992.

[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] Digital Equipement Corp. *DIGITAL JTrek* . http://www.digital.com/java/download/jtrek/index.html.

[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] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1997.

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

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

[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] F. B. Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, 1998.

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

[19] R. Tolksdorf. Programming languages for the Java Virtual Machine. Available from http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html.