Formal Methods in System Design, 24, 129–155, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Java-MaC: A Run-Time Assurance Approach for Java Programs*

MOONZOO KIM moonjoo@secui.com; moonzoo@postech.co.kr
Department of Computer Science and Engineering, Pohang University of Science and Technology, Korea

MAHESH VISWANATHAN vmahesh@cs.uiuc.edu
Department of Computer Science, University of Illinois at Urbana-Champaign, USA

SAMPATH KANNAN kannan@saul.cis.upenn.edu
INSUP LEE lee@saul.cis.upenn.edu
OLEG SOKOLSKY sokolsky@saul.cis.upenn.edu
Department of Computer and Information Science, University of Pennsylvania, USA

Abstract. We describe Java-MaC, a prototype implementation of the Monitoring and Checking (MaC) architecture for Java programs. The MaC architecture provides assurance that the target program is running correctly with respect to a formal requirements specification by monitoring and checking the execution of the target program at run-time. MaC bridges the gap between formal verification, which ensures the correctness of a design rather than an implementation, and testing, which does not provide formal guarantees about the correctness of the system.

Use of formal requirement specifications in run-time monitoring and checking is the salient aspect of the MaC architecture. MaC is a lightweight formal method solution which works as a viable complement to the current heavyweight formal methods. In addition, analysis processes of the architecture including instrumentation of the target program, monitoring, and checking are performed fully automatically without human direction, which increases the accuracy of the analysis. Another important feature of the architecture is the clear separation between monitoring implementation-dependent low-level behaviors and checking high-level behaviors, which allows the reuse of a high-level requirement specification even when the target program implementation changes. Furthermore, this separation makes the architecture modular and allows the flexibility of incorporating third party tools into the architecture. The paper presents an overview of the MaC architecture and a prototype implementation Java-MaC.

Keywords: software reliability, formal specification, run-time monitoring and checking, execution trace validation, program instrumentation, Java, Java bytecode engineering

1. Introduction

In the past two decades, much research has concentrated on the methods for analysis and validation of software systems, which are deployed in safety critical areas such as avionics and automobiles. Many successful industrial case studies have been conducted in the area of formal verification [5]. Complete formal verification, however, has not yet become a prevalent analysis method. Reasons for this are twofold. First, complete verification of

*This research was supported in part by NSF CCR-9988409, NSF CCR-0086147, NSF CCR-0209024, ARO DAAD19-01-1-0473, and ONR N00014-97-1-0505.

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 software engineers 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 is hard to guarantee whether or not the current execution of the system is correct using the two traditional methods: verification and testing. Therefore, the approach of continuously monitoring and checking a running system with respect to formal requirements specifications can be used to fill the gap between these two approaches. This approach might not seem very useful at first glance because detecting errors does not seem interesting; for example, just reporting that a system is about to crash is not helpful. Run-time monitoring, however, could help users of the system to detect and correct errors. First, subtle errors are hard to detect without thorough run-time monitoring. Second, errors may not cause disastrous system failure immediately. Run-time checking can find such errors in a timely manner and help the users to take recovery actions before critical failure happens.

In this paper, we describe *Java-MaC*, a run-time assurance system for Java programs based on the Monitoring and Checking (MaC) architecture. The objective of the MaC architechture is to provide assurance that the target program is running correctly with respect to a formal requirements specification. The use of formal requirements specifications in run-time monitoring and checking is the salient aspect of the MaC architecture. Furthermore, monitoring and checking as well as target program instrumentation are automatically performed from a given requirement specification. These automatic processes based on requirement specifications make the run-time analysis rigorous. Another characteristic feature of the architecture is the separation between monitoring program-dependent low-level behavior and checking high-level behavioral requirements (see figure 1). This separation allows the specification of high-level requirements independent of the implementation since implementation specific details are confined to the low-level specification. In addition, this separation enables the reuse of a high-level requirements specification even when the target program implementation changes. Furthermore, this modularity of the MaC architecture and its implementation based on well-defined interfaces among the components makes it easy to incorporate third-party tools into the architecture. For example, we were able to link the Java-MaC system with a network simulator to analyze the correctness of network simulation traces [3].

The MaC architecture is a general framework, which is not limited to any specific programming language. To demonstrate its effectiveness, however, we have implemented a MaC prototype for Java programs, called *Java-MaC*. Java-MaC targets Java executable code (i.e., bytecode). It is easy to deploy Java-MaC, because it automatically instruments

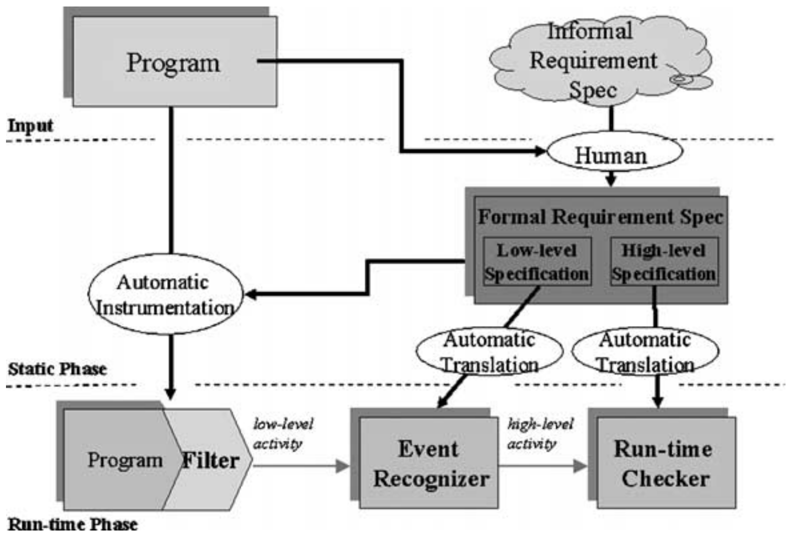


Figure 1. Overview of the MaC architecture.

the target program and generates the run-time components of Java-MaC based on requirements specifications written in two scripting languages.

The paper is organized in two parts. The first part, Sections 2 and 3, briefly describes the MaC architecture. Section 2 presents an overview of the MaC architecture. Section 3 presents the languages for requirements specification. The second part—Section 4 to Section 7—focuses on Java-MaC. Section 4 discusses issues on monitoring Java programs. Section 5 describes the Java-MaC implementation. Section 6 describes overhead reduction techniques used in Java-MaC. Section 7 provides a small but illustrative example of a stock client program. Section 8 presents related work. Finally, Section 9 summarizes and concludes the paper.

2. Overview of the MaC architecture

The overall structure of the architecture is shown in figure 1. The architecture includes two main phases: a *static phase* and a *run-time phase*. During the static phase (i.e., before a target program runs), the run-time components namely a *filter*, an *event recognizer*, and a *run-time checker* are automatically generated from a formal requirements specification. During the run-time phase, (i.e., while the target program executes), information about the execution of the target program is collected and checked against the given formal requirements specifications.

2.1. Static phase

The static phase of the MaC architecture starts with a formal requirements specification. A formal requirements specification is written in two separate parts: a high-level specification and a low-level specification. A high-level specification consists of required properties. A low-level specification contains the definitions of primitive events and conditions used in the high-level specification. These definitions are in terms of program entities such as program variables and program methods, and their purpose is to assign high-level meanings to the program entities. For example, in a gate controller of a railroad crossing system, the safety requirements that the gate must be completely down when a train is in the crossing may be expressed using the condition `InCrossing`. The target program, on the other hand, stores the position of the train in a variable `train_position`. Here, the low-level specification can define the condition `InCrossing` as `(600 < train_position) && (train_position < 800)`, where the starting position of the crossing is 600 and the ending position of the crossing is 800.

The separation of a low-level specification and a high-level specification has two benefits. First, different implementations can be monitored using the same high-level specification since only the low-level specification needs to be modified according to the new implementation. Second, requirements are more naturally specified since implementation specific details are abstracted.

Once the specifications are written, the next task is to generate run-time components. A filter is generated from the low-level specification and inserted into the target program through the automatic instrumentation procedure. An event recognizer is generated also from the low-level specification automatically. Similarly, a run-time checker is generated automatically from the high-level specification.

2.2. *Run-time phase*

During the run-time phase, the instrumented target program is executed while being monitored and checked with respect to a requirements specification.

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

3. The MaC language

In this section, we give a brief overview of the languages used to describe specifications. The language for low-level specification is called Primitive Event Definition Language (PEDL). PEDL is used to define what information is to be sent from the filter to the event recognizer, and how it is transformed into events used in the high-level specification by the event recognizer. High-level specifications are written in the Meta Event Definition Language (MEDL). One could conceivably write all the correctness requirements in a high-level programming language such as Java or C, by directly coding the monitor in these languages. However, using a specification language like MEDL has the advantage that the user can specify the correctness requirements "declaratively," without worrying about the "operational" details of the monitor. This can help users build reliable and correct monitors for properties more easily. This separation ensures that the architecture is portable to different implementation languages and specification formalisms. In Section 3.1, we introduce the distinction between *events* and *conditions*, that is fundamental to the MaC architechture. 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).

Before presenting the two languages, we first define the notions of *event* and *condition*, which are fundamental to the MaC architecture.

3.1. Events and conditions

The MaC architecture assumes that it is possible to observe the behavior of the target system and evaluate the observed behavior to check whether required properties are satisfied or not. The observed behavior consists of "interesting" state changes in the target system. We distinguish between two kinds of interesting state changes—events and conditions.

Events occur instantaneously during the system execution, whereas *conditions* represent information that holds 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` has the value 2. The distinction between events and conditions is very important in terms of what the monitor can infer about the execution based on the information it gets from the filter. The monitor can conclude that an event does not occur at any moment except when it receives an update from the filter. By contrast, once the monitor receives a message from the filter that variable `position` has been assigned the value 2, we can conclude that `position` retains this value until the next update.

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 later when we present the logic.

We have two attributes `time` and `value`, defined for events. `time(e)` gives the time of the last occurrence of event `e`. `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 properly synchronized to simplify the presentation of this paper. In addition, an event can have an attribute value. `value(e)` gives the value associated with `e`, provided `e` occurs.

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 interpreting logical formulae having undefined variables. The problems associated with defining the semantics of logics in the presence of partial functions[1] are well-understood [4, 9, 21]. 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 do not work when the logic has 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 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 and 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, 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. Figure 2 shows the syntax of conditions (C) and events (E).

$$\begin{array}{lcl}
\langle C\rangle & ::= & c \mid \texttt{defined}(\langle C\rangle) \mid [\langle E\rangle,\langle E\rangle) \mid !\langle C\rangle \\
 & & \mid \langle C\rangle\&\&\langle C\rangle \mid \langle C\rangle||\langle C\rangle \mid \langle C\rangle\Rightarrow\langle C\rangle \\
\langle E\rangle & ::= & e \mid \texttt{start}(\langle C\rangle) \mid \texttt{end}(\langle C\rangle) \mid \langle E\rangle\&\&\langle E\rangle \\
 & & \mid \langle E\rangle||\langle E\rangle \mid \langle E\rangle \ \texttt{when} \ \langle C\rangle
\end{array}$$

Figure 2. The syntax of conditions and events.

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 primitive 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 $\{\mathit{true}, \mathit{false}, \Lambda\}$, 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 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. 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 4.[2]

As stated before, we 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 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

[c_k primitive] $\mathcal{D}^t_M(c_k) = L_C(s_i, c_k)$, where $\tau(s_i) \leq t$ and for all s_j $(j > i)$ $\tau(s_j) > t$

[defined]
$$\mathcal{D}^t_M(\text{defined}(c)) = \begin{cases} \mathit{true} & \text{if } \mathcal{D}^t_M(c) \neq \Lambda \\ \mathit{false} & \text{otherwise} \end{cases}$$

[pair]
$$\mathcal{D}^t_M([e_1, e_2]) = \begin{cases} \mathit{true} & \text{if there exists } t_0 \leq t \text{ such that} \\ & M, t_0 \models e_1 \\ & \text{and for all } t_0 \leq t' \leq t, \\ & M, t' \not\models e_2 \\ \mathit{false} & \text{otherwise} \end{cases}$$

[negation]
$$\mathcal{D}^t_M(!c) = \begin{cases} \mathit{true} & \text{if } \mathcal{D}^t_M(c) = \mathit{false} \\ \Lambda & \text{if } \mathcal{D}^t_M(c) = \Lambda \\ \mathit{false} & \text{if } \mathcal{D}^t_M(c) = \mathit{true} \end{cases}$$

[disjunction]
$$\mathcal{D}^t_M(c_1 \| c_2) = \begin{cases} \mathit{true} & \text{if } \mathcal{D}^t_M(c_1) \text{ or } \mathcal{D}^t_M(c_2) \text{ is } \mathit{true} \\ \mathit{false} & \text{if } \mathcal{D}^t_M(c_1) = \mathcal{D}^t_M(c_2) = \mathit{false} \\ \Lambda & \text{otherwise} \end{cases}$$

[conjunction] $\mathcal{D}^t_M(c_1 \&\& c_2) = \mathcal{D}^t_M(!(!c_1 \| !c_2))$

[implication] $\mathcal{D}^t_M(c_1 \Rightarrow c_2) = \mathcal{D}^t_M(!c_1 \| c_2)$

Figure 3. Denotation for conditions.

$M, t \models c$	iff	$\mathcal{D}^t_M(c) = \mathit{true}$
$M, t \models e_k$ (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 if $i > 0$ then $M, \tau(s_{i-1}) \not\models c$; i.e., start$(c)$ occurs when condition c changes from false or undefined, to true; before state s_0 conditions are assumed to be undefined.
$M, t \models \text{end}(c)$	iff	$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i) \models !c$ and if $i > 0$ then $M, \tau(s_{i-1}) \not\models !c$; i.e., end$(c)$ occurs when condition c changes from true or undefined, to false; before state s_0 conditions are assumed to be undefined.
$M, t \models e_1 \| e_2$	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 4. Semantics of events and conditions.

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". Also, observe that MaC reasons about temporal behavior and data behavior of the target program execution using events and conditions; events are an abstract representation of time and conditions are abstract representation of data.

3.4. Primitive event definition language (PEDL)

PEDL is the language for writing low-level specifications. The design of PEDL is based on the following two principles. First, we encapsulate all implementation-specific details of the monitoring process in PEDL specifications. 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 specifications is to define primitive events of requirement specifications. The operations on events can be used to construct more complex events from these primitive events. PEDL is dependent on its target programming language. We will describe PEDL for Java in Section 4.1.

3.5. Meta event definition language (MEDL)

The safety requirements are written in MEDL. Primitive events and conditions in MEDL specifications are imported from PEDL specifications; hence the language has the adjective “meta”. The overall structure of a MEDL specification is given in figure 5.

Importing events and conditions. A list of events and conditions to be imported from an event recognizer is declared.

Defining events and conditions. Events and conditions are defined using imported events, imported conditions, and auxiliary variables, whose role is explained later in this section.

```
ReqSpec <spec_name>

  /* Import section */
  import event <e>;
  import condition <c>;

  /*Auxiliary variable declaration*/
  var int <aux_v>;

  /*Event and condition definition*/
  event <e> = ...;
  condition <c>= ...;

  /*Property and violation definition*/
  property <c>  = ...;
  alarm <e> = ...;

  /*Auxiliary variable update section*/
  <e> -> {  <aux_v'> := ... ; }
End
```

Figure 5. Structure of MEDL.

These events and conditions are then used to define safety properties and alarms.

Safety properties and alarms. The correctness of the system is described in terms of safety properties and alarms. Safety properties are conditions that must be *always* true during the execution. Alarms, on the other hand, are events that must never be raised (all safety properties [19] can be described in this way). Also observe that alarms and safety properties are complementary ways of expressing the same thing. The reason that we have both of them is because some properties are easier to think of in terms of conditions, while others are in terms of alarms.

Auxiliary variables. The language described in Section 3.1 has a limited expressive power. For example, one cannot count the number of occurrences of an event, or talk about the *i*th occurrence of an event. For this purpose, MEDL allows users to define auxiliary variables, whose values may then be used to define events and conditions. Updates of auxiliary variables are triggered by events. For example,

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

counts occurrences of event `e1`. A special auxiliary variable `currentTime` is used to refer to the current time of the target program. It is set to be the last timestamp received from the filter.

4. Monitoring Java programs

Our approach to monitoring the execution of a Java program is to insert instrumentation code within Java bytecode. The instrumentation code is executed as part of the execution of a Java program to collect necessary state information. There are two conflicting objectives for supporting the monitoring of Java programs. On the one hand, it is necessary to ensure that all information relevant to check the required properties is collected during the program execution. On the other hand, information extraction should not cause undue overhead on the running program.

A Java program is a collection of objects. Each object has an internal state that is a collection of object fields and a set of methods that can be invoked during execution. Methods, in addition, may have local variables. Object fields and local variables may be references to other objects; however, the state of the program is ultimately contained in the fields and variables of primitive types. Based on this observation, Java-MaC limits the monitoring to the values of fields and local variables of primitive types and to method calls. Monitored entities are declared and used to define events and conditions in a monitoring script written in PEDL for Java, called Java-PEDL.

This section describes Java-PEDL in detail. Then, we discuss issues associated with monitoring objects when objects are aliased, that is, multiple references may be used to update the same field of the same object. Finally, we discuss how to instrument Java bytecode according to Java-PEDL scripts.

4.1. PEDL for Java

By the design of the MaC architecture, PEDL is closely related to the target programming language because events are defined using program entities such as variables and methods. The automatic instrumentation provided by Java-PEDL must preserve the functional correctness of the target program and allow fast recognition of events. This means that although Java-MaC has side effects on resource consumption such as memory and CPU usage, no program variables are modified by the instrumentation code. In addition, Java-MaC does not change the control flow of the program unless the program has synchronization errors. Efficiency of event recognition is achieved by not allowing recursive or circular expressions in the definitions of events and conditions. We believe that the current design of Java-PEDL provides the right balance between the expressive power of the language and efficiency of implementation.

The overall structure of a Java-PEDL specification is shown in figure 6. Each specification consists of four sections: export declaration, overhead reduction flag, monitored entity declaration, and event and condition definition.

Exporting events and conditions. This section declares the list of events and conditions to be exported from the event recognizer to the run-time checker. These events and conditions are defined in the event and condition definition section.

Overhead reduction flags. The user can enable or disable various overhead reduction techniques. We discuss these techniques in Section 6.

```
MonScr <spec_name>
  /* Export section */
  export event <e>;
  export condition <c>;

  /* Overhead reduction section */
  [timestamp;]
  [valueabstract;]
  [deltaabstract;]
  [multithread;]

  /* Monitored entity declaration section */
  monobj <var>;
  monmeth <meth>;

  /* Event and cond definition */
  event <e> = ...;
  condition <c>= ...;
End
```

Figure 6. Structure of PEDL for Java.

Declaring monitored variables and methods. Each declaration identifies an object that needs to be monitored. This object resides in a memory location. Since the exact memory location of the object is not known during the static phase, this object is specified in a monitoring script as a chain of references that starts in a fixed place in the object graph of the Java program: either a static variable of a class or a local variable of a static method such as `main()`. That is, when we specify such a chain of references in a monitoring script, it effectively becomes a name for the memory location of the monitored object. In order to maintain the correspondence between this name and the object location, we assume that references in the chain used in the declaration of the monitored object, once assigned, do not change.

Java-PEDL allows the monitoring of fields and variables of the primitive types, but does not allow objects to be monitored directly. This restriction is adopted deliberately to minimize monitoring overhead for the following two reasons. First, the overhead of monitoring an object can be quite significant since if an object contains references to other objects, then every change to any node that is recursively referenced needs to be detected. Second, when we detect that an object has changed (i.e., some node in the object graph has changed), the whole object graph may need to be delivered to the event recognizer. We note that monitoring primitive variables only, but not objects, is not a severe restriction because primitive variables constitute and define an object. For the rest of the paper, whenever we say monitoring an object, we mean monitoring primitive variables of the object.

Java-PEDL identifies *execution points* to be monitored. There are two possible ways to identify execution points in a Java program: source code line numbers and the beginnings and endings of methods. Java-PEDL allows the use of starts and ends of methods as monitored execution points, but not source code line numbers. This is because a line number does not have inherent meaning in the target program; for example, reformatting a single statement to span over two lines can change line numbers. Furthermore, source code may not be available at the point at which instrumentation is performed.

Defining events and conditions. Primitive conditions in Java-PEDL are constructed from boolean-valued expressions over the monitored variables. An example of such a condition is

```
condition InCrossing = (600 < Train.position) &&
                       (Train.position < 800)
```

In addition to these conditions, Java-PEDL also supports the primitive condition of the form `InM(f)`, where `f` is the name of a method. This condition is true as long as execution is currently within the method `f`. Complex conditions are built from primitive conditions using boolean connectives.

Primitive events in Java-PEDL correspond to updates of monitored variables and calls/returns of monitored methods. The event `update(x)` is triggered when the variable x is assigned a value. Events `startM(f)` and `endM(f)` are triggered when control enters the method `f` and returns from `f`, respectively. For example,

```
event OpenGate = startM(Control.open())
```

defines an event meaning a controller starts opening a gate. The operations on events in figure 3 can be used to construct more complex events from these primitive events. As stated earlier, Java-PEDL forbids the use of recursion or circularity in the definition of these complex events.

For each event, Java-PEDL defines two attributes, `time` and `value`. `time(e)` gives the time of the last occurrence of event `e`. `value(e,i)` gives the ith value in the tuple of values associated with `e`, provided `e` occurs. Java-PEDL defines the values of primitive events as follows. The value of `update(var)` is the current value of `var`. The value of `startM(method)` is a tuple containing parameters of the `method` when the `method` starts. The value of `endM(method)` is a tuple containing the parameters of the `method` and a return value (if any) when the `method` ends. The value of (`e when cond`) is the value of the event `e`.

4.2. Monitoring objects

A Java program forms a complex object graph. Java handles an object via a *reference* pointing to the object. Since many references can point to the same object, the same object can be updated by different object references. For example, consider the variable `x` pointed to by an arrow in figure 7. It can be referenced by `a.b2.x` and `a.b1.b'.x`. Therefore, if the monitored object was declared as `a.b2.x`, then it is not enough to monitor updates to the variable `x` by `a.b2.x`.

There are two issues. The first is how to identify a variable within an object initially. Java-PEDL requires that a variable be specified with its full path in the object graph. This allows, for example, a variable `x` in the object graph such as `a.b2.x` to be distinguished from a variable `x` in a different instance of the same object type such as `a.b1.x`.

The second issue is how to detect all the updates to the variable, `a.b2.x`, regardless of how it is referenced. In order to do this, we need to keep track of the addresses of objects that contain monitored variables. For this, Java-MaC maintains a globally accessible table containing addresses of monitored objects and monitored object names, called *address table*. The address table resides in the filter. Since an object is located at a unique address in the heap, comparing the addresses of two objects allows us to efficiently distinguish one object from another. This also allows us to detect all updates to a monitored object field, no

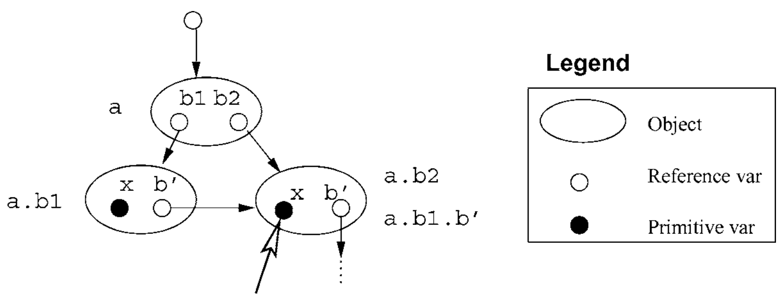


Figure 7. An object graph.

matter through what chain of references these updates happen. The use of an address table reinforces the principle put forth in Section 4.1, that a monitored object corresponds to a fixed memory location.

The current implementation of the address table assumes however that there is no aliasing in the *declarations* of the monitored objects. If, in the example above, both `a.b2.x` and `a.b1.b'.x` were declared as monitored objects, they would correspond to the same memory object and require that two messages be sent to the event recognizer for every update to that object. We have disallowed such aliasing to avoid the extra overhead.

4.3. Instrumentation process

Java-MaC monitors *field primitive variables*, *local primitive variables*, and *starts/ends of methods*. The Java-MaC instrumentor detects instructions which update monitored variables or instructions located at the beginnings/endings of methods.

- *Field primitive variables.* Static field variables and instance field variables are updated by `putstatic` and `putfield` bytecode instructions, respectively. For example, `putstatic A/x I` updates a static variable `x` of integer type (`I`) declared in a class `A` with the top element in the operand stack.
- *Local primitive variables.*[3] Local primitive variables are updated by `<T>store`, `<T>store_<n>` and `iinc` where `<T>` is a variable type and `<n>` $\in \{1, 2, 3, 4\}$ which indicates `<n>`th local variable. For example, `istore x` in a method `m` updates a local variable `x` declared in `m` with the top element in the operand stack.
- *Execution points.* There is only one starting point in a method—the beginning of a method definition. There can however be several ending points—the locations of `return` instructions. The parameters and the return variable of a method are monitored as local variables of the method.

During the static, compile phase, the Java-MaC instrumentor identifies candidate update instructions for the monitored variables. Once the instrumentor recognizes a candidate update instruction (say i) for a monitored variable, `varName`, the instrumentor inserts a probe consisting of `monitorenter Filter.lock` and `sendObjMethod(Object parentAddress, <T> value, String varName)` right before i, where `parentAddress` is an address of an object whose field `varName` is monitored. At runtime, when `sendObjMethod()` is called, it checks whether or not a variable this probe monitors is actually a monitored variable by matching `parentAddress` with the address of a monitored object in the address table. If the variable is a monitored variable, `sendObjMethod()` sends it to the event recognizer; otherwise, nothing is sent.

The instrumentor inserts `monitorexit Filter.lock` right after i. The pair of `monitorenter` and `monitorexit` ensures that the update to a variable and the sending of its new value are executed atomically (see Section 5.2.1). For execution points, the instrumentor inserts probes at the starting point of a method (i.e., the beginning of a method definition) and at the ending points of a method (i.e., locations where `return` instructions exist).

```
01: aload_1
02: aload_0
03: getfield B/y I
04:   getstatic Filter/lock Ljava/lang/Object
05:   monitorenter
06:   dup2
07:   ldc ''x''
08:   invokestatic
      SendMethods/sendObjMethod(Ljava/lang/Object;ILjava/lang/String;)V
09: putfield A.b I
10:   getstatic Filter/lock Ljava/lang/Object
11:   monitorexit
```

Figure 8. Inserted probe in bytecode (indented lines are the probe).

As an example, figure 8 illustrates the probes that Java-MaC inserts to monitor a variable `int a.b`, where `a` is of type `A`. Indented lines are the probes inserted by Java-MaC. Line 9 updates the field `x` of class `A` with the value of the field `y` of class `B`. This line is identified during the instrumentation process as potentially affecting the monitored variable `a.b`, and the probe is inserted in front of it. Lines 4 and 5 acquire the lock to ensure that the update and its monitoring happen atomically. Line 6 duplicates the value to be stored, which was obtained by the instruction in Line 3, on the stack. Line 7 adds the name of the field as the second argument. Line 8 invokes `sendObjMethod()` with the new value of `A.b`. Lines 10 and 11 release the lock after `A.b` is updated.

5. The MaC prototype for Java

This section describes the MaC prototype for Java programs, called *Java-MaC*. Figure 9 shows the overall structure of Java-MaC, which is divided into the static phase and the

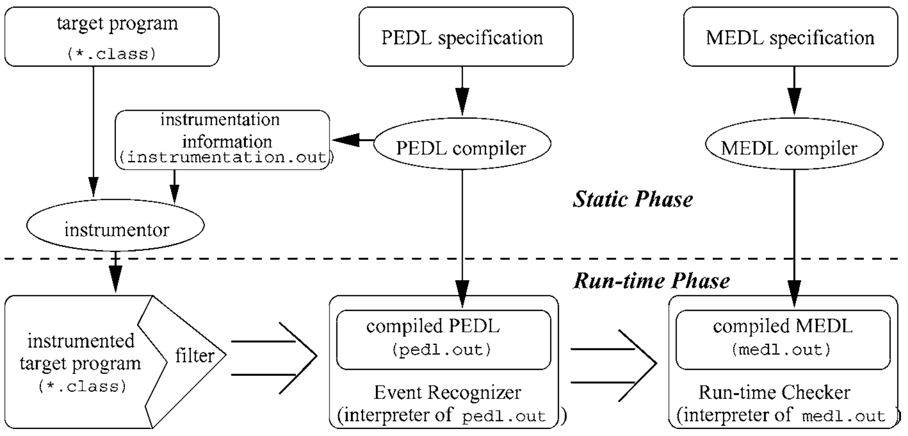


Figure 9. Java-MaC.

dynamic phase. Section 5.1 describes the Java-MaC components of the static phase. Section 5.2 describes the run-time components of Java-MaC.

5.1. The static phase

Java-MaC has three static-phase components: an *instrumentor*, a *PEDL compiler*, and a *MEDL compiler*. The PEDL compiler compiles the PEDL script into an abstract syntax tree, which is evaluated by the event recognizer at run-time. At the same time, the PEDL compiler generates instrumentation information, which is used by the instrumentor. The Java-MaC instrumentor takes as input a collection of Java class files in the bytecode format and instrumentation information that contains a list of monitored variables, monitored methods and monitoring flags generated from a PEDL script. Based on these two inputs, the Java-MaC instrumentor inserts a filter into the target bytecode. Similarly, the MEDL compiler compiles the MEDL script into an abstract syntax tree (`medl.out`), which is evaluated by a checker at run-time. The instrumentor and PEDL/MEDL compilers are detailed in [14].

5.2. The run-time phase

The Java-MaC run-time components consist of a *filter*, an *event recognizer*, and a *run-time checker*. A detailed description of each component is given below.

5.2.1. The filter. The purpose of the filter is to monitor the execution of the target program, and extract snapshots of the program execution that are useful to the event recognizer. The filter consists of the following two functional parts:

- *Probes*: Probes are inserted into *all* locations where monitored variables are updated and where monitored methods begin and end. For a monitored variable, probes extract the new value of the variable, and pass it to the filter thread. For a monitored method, probes detect when execution enters and exits the method, and pass the information to the filter thread.
- *Filter thread*: A target program is not originally designed to communicate with an event recognizer. Communication from the target program to the event recognizer is carried out by the filter thread. Probes communicate with the filter thread using a buffer. The filter thread sends the contents of the buffer to the event recognizer through a communication channel.

It is important that the filter reports the monitored updates in the same order as they happen in the program. If several threads update the same monitored variable, the variable update and the corresponding report delivery need to happen atomically. To achieve atomicity, the filter provides a globally accessible lock that all threads acquire before updating a monitored variable and release after the report is complete. Figure 10 shows the structure of a filter with two threads performing concurrent updates.

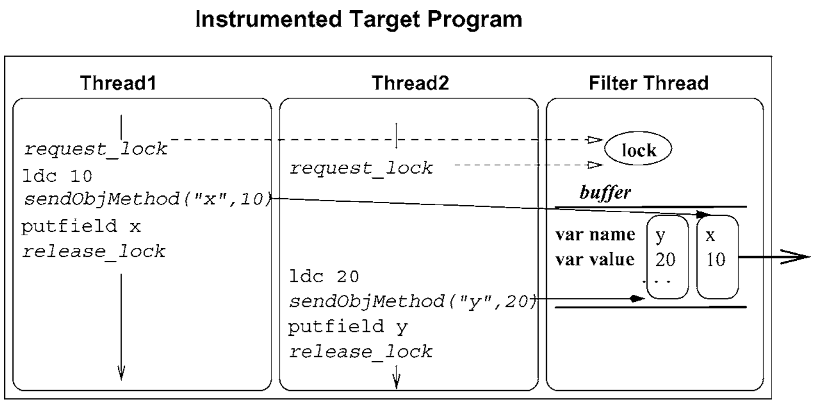


Figure 10. Structure of a filter.

The overhead of locking can be non-trivial. However, we observe that locking is unnecessary if a monitored variable is updated by one thread only. In Java-MaC, the use of locking in the filter is optional. Java-PEDL provides a keyword `multithread` in the overhead reduction section (see Section 4.1). Locks are used by probes only when this keyword is present in the monitoring script. Although this `multithread` flag should be specified for each variable separately, the current Java-MaC only supports one flag that applies to all the monitored variables.

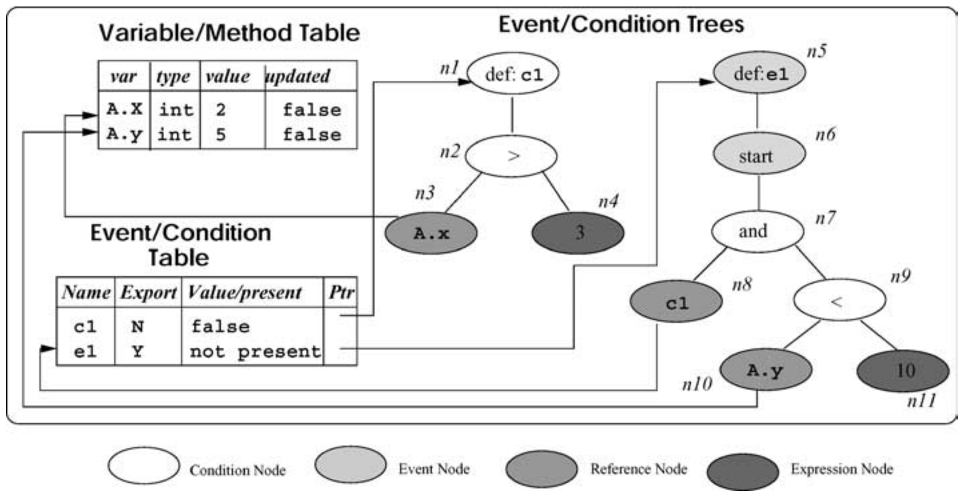
5.2.2. The event recognizer. From a PEDL script, the PEDL compiler generates `pedl.out` which consists of a set of abstract syntax trees and two tables. One table contains the names and the current values of monitored variables and methods. The other table lists the names of events and conditions, and for each name, it stores a reference to a respective event or condition tree. The set of abstract trees captures the definitions of events and conditions as well as the dependencies between events and conditions specified in a monitoring script.

As an example, figure 11 shows a simple PEDL script and figure 12 shows a graphical representation of the `pedl.out` of that script.

```
MonScr test
  export event e1;
  monobj int A.x;
  monobj int A.y;

  condition c1 = A.x > 3;
  event e1 = start(c1 && A.y < 10);
end
```

Figure 11. A simple PEDL script.



var	*type*	*value*	*updated*
A.X	int	2	false
A.y	int	5	false

Name	*Export*	*Value/present*	*Ptr*
c1	N	false	
e1	Y	not present	

Figure 12. `pedl.out` of the PEDL script in figure 11.

```
1.   Clear all the marks of the roots of event/condition trees
2.   For each event/condition tree
2.1. Set the mark of the root
2.2. Evaluate children
       – For an operator/connective node, get the value of the node by
         performing corresponding operation on the values of the
         childrenwhich are obtained by repeating Step 2.2
       – For a reference node for a variable/method, get the value from
         variable/method table
       – For a reference node for an event/condition, get the present
         flag/truth value of the node
         from the root of the corresponding event/condition tree
         • if the root has mark set, record the flag/value of the root
         • if not, repeat Steps 2.1 and 2.2
3.   Send exported events whose present flags are set and exported
     conditions whose old values and new values are different
```

Figure 13. Evaluation of a dependency graph.

Whenever the event recognizer receives a new snapshot from the filter, it reevaluates the events and conditions. Figure 13 describes the algorithm used by the event recognizer to evaluate events and conditions defined in `pedl.out`. Each root of an abstract tree has a marker used by the algorithm. In the events and conditions table, each event has a present-flag entry that is set if an event is present now, whereas each condition has a truth-value entry that is true if the condition is true and false otherwise.

We now illustrate the evaluation algorithm using the example above. Suppose that the event recognizer has `A.x` as 2 and `A.y` as 5. Then, suppose the filter sends a snapshot of `A.x` as 5. When the event recognizer receives the snapshot, it first updates `A.x` as 5 in the value/method table. Then, it removes marks on all roots of event/condition trees and starts evaluating events and condition expressions from the roots in a top-down manner.

The order of evaluating event/condition trees does not affect the evaluation result. In other words, whether the event recognizer starts evaluation from $n1$ or $n5$ does not change the result of evaluation. Suppose that $n1$ is evaluated first. First, the mark of $n1$ is set. Then, the truth value field of $n1$ (false) is copied into its old truth value field and the value of $n1$ changes to *true* because the new value, 5, of `A.x` is greater than 3. Although `c1` has changed its value, `c1` need not be sent to the checker because it is not declared as exported. Next, $n5$ is evaluated. When $n8$ is evaluated, the event recognizer recognizes that $n1$ has been already evaluated by looking at the mark of $n1$. Thus, the event recognizer does not evaluate $n1$ again, but just uses the value of $n1$. Finally, $n5$ is evaluated as present because $n8$ becomes true, and thus, $n7$ changes from false to true. Since `e1` is declared as exported, `e1` is sent to the run-time checker.

PEDL expressions are evaluated in time that is linear in the size of the expressions. This is because the PEDL dependency graph is acyclic and the evaluation algorithm is essentially performing a topological sort of this graph.

5.2.3. Run-time checker. The run-time checker evaluates event and condition definitions in the abstract syntax tree in `medl.out` whenever the run-time checker receives events or conditions from the event recognizer. MEDL expressions are evaluated in time linear to the size of expression, because, like PEDL, MEDL does not allow recursive expressions. If the run-time checker detects a violation defined by `alarm` or `property`, it raises a signal. The evaluation procedure is similar to that of an event recognizer except for additional steps for auxiliary variable updates. More details are in [23].

6. Overhead reduction

Any monitoring approach that does not use specialized hardware, causes overhead to the target system. In this section, we analyze the overhead imposed on the target system by instrumentation and communication between the target system and the monitor. We then suggest three techniques to reduce this overhead without compromising the correctness of monitoring, which can be specified using overhead reduction flags, `timestamp, deltaab-stract, valueabstract`, in PEDL as shown in figure 6.

Our overhead analysis is based on the model of the monitoring process shown in figure 14. The model captures overheads in the filter and the monitor. The monitor combines the event recognizer and the checker, and thus, this model ignores the effect of the internal communication between them within the monitor. The following six parameters constitute overheads in the model. p is the overhead of executing a probe. w is the overhead of writing a snapshot to the buffer. *send* is the overhead of sending the content of the buffer to the monitor when the buffer becomes full. *Rec* is the overhead of receiving snapshots into the buffer of the monitor. r is the time taken to read a snapshot from the buffer. e is the overhead of evaluating properties upon arrival of a snapshot. We will use P, W, *Send*, *Rec*, R, and E to refer to the accumulated values of p, w, *send*, *rec*, r, and e, respectively, during the target program execution.

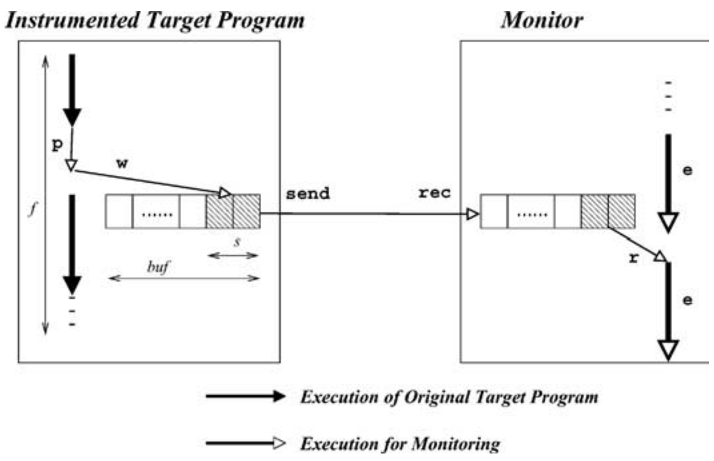

Instrumented Target Program
Monitor
p
w
f
send
rec
s
buf
e
r
e
Execution of Original Target Program
Execution for Monitoring

Figure 14. The model of monitoring overheads to a target system.

Our discussion is limited to the overhead reduction techniques that can be applied by the filter. In particular, we are interested in techniques that can be specified in PEDL. There are three factors that we can control to reduce the overheads at the filter—the snapshot size s, the frequency of taking snapshots f, and the buffer size *buf*.

1. Reducing the snapshot size s decreases W, R, *send* and *Rec*.
2. Reducing the snapshot frequency f decreases P, W, R, and E. Since the volume of snapshots delivered from the target program to the monitor is decreased, this also reduces the communication overheads *Send* and *Rec*.
3. Increasing the buffer size *buf* decreases *Send* and *Rec* if each send and receive operation has high startup overhead.

Reducing s or f may compromise the correctness of property evaluation, since we either omit some information in the snapshot or do not send all snapshots. On the other hand, increasing *buf*, does not affect the correctness of property evaluation although increased *buf* delays the delivery of snapshots which may result in late detection of violations. We present two techniques for reducing s and f correctly.

6.1. Reducing snapshot size

In Java-MaC, a snapshot sent from the filter to the event recognizer consists of an ID of the monitored variable (1 byte) and the value of the variable (from 1 to 8 bytes). Since each timestamp is 8 bytes long, if we attach a timestamp to every snapshot, the timestamp would take a large portion (between 47% and 80%) of the total space taken for the snapshot. To reduce the snapshot size, it is possible to send timestamps periodically instead of attaching a timestamp to every snapshot. The timing inaccuracy caused by such a periodic time stamp is bounded by the period used. Note that periodic timestamps need to contain an ID for timestamp, but not the actual time value since it only needs to signal the beginning of the next period. When snapshots are frequently exported, the savings from such periodic timestamps can be quite significant. Suppose the period is 100 milliseconds and a snapshot is sent every millisecond. With a timestamp attached to every snapshot, the timestamps take the total 8000 bytes per second. Periodic timestamps, however, take only 10 bytes per second. This type of reduction can be specified in Java-PEDL by using the `timestamp` keyword and passing the timestamp delivery period as a parameter to the instrumentor.

Another way to reduce the snapshot size is to use a *delta* value. When a difference (called delta value) between two consecutive values of a monitored variable is small enough to be represented in smaller data type (*delta* type), the filter can send delta value instead of new value. When a snapshot has a delta value instead of a normal value, an ID field has a negative number to indicate use of a delta value. The delta type of `int` and `short` is `byte`. The delta type of `long` is `int`. The delta type of `double` is `float`. This option is specified by including `deltaabstract` in Java-PEDL.

6.2. Reducing sampling rate

Not every update of a variable affects requirement properties. Therefore, a filter may send only snapshots which affect properties. We call this technique *value abstraction*. Value abstraction, however, can require expensive computation by the filter thread. Value abstraction also violates the separation of concerns that we have carefully established between the filter and the other components of the monitor. Thus, we apply this technique only to cases where we can check whether properties are affected or not using a simple, fast test.

A simple expression $sexp_x$ is defined as "x *cmp* c," where x is a monitored variable, *cmp* is one of $>$, $>=$, $==$, $<=$, $<$, and c is a constant. Evaluating a simple expression can be much more efficiently compared to the overhead of sending a value and then evaluating the event or condition scripts that depend on the value. If `x` appears only in $sexp_x$'s, Java-MaC applies value abstraction to the variable x (i.e., exports the value of x only when any of $sexp_x$'s changes). $sexp_{xi}$ are obtained from the event or condition definition of a PEDL script. We illustrate how simple expressions are obtained from condition definitions by considering the following two conditions:

```
condition c1 = (3 < x && x < 10) || y >10 || z > 10;
condition c2 = x > 5 && w > 2*z + 3;
```

From these, the following five simple expressions are identified:

$$sexp_{x0} = x > 3$$
$$sexp_{x1} = x < 10$$
$$sexp_{y0} = y > 10$$
$$sexp_{z0} = z > 10$$
$$sexp_{x2} = x > 5$$

x appears only in $sexp_{x0}$, $sexp_{x1}$, and $sexp_{x2}$ in the PEDL script. Whenever x is updated, the probe checks whether any of the value of $sexp_{xi}$ has changed. If the value of any $sexp_{xi}$

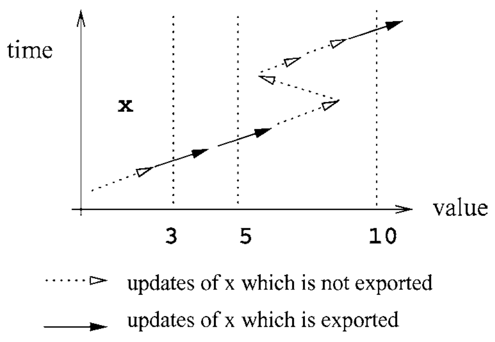

time
x
value
3
5
10
updates of x which is not exported
updates of x which is exported

Figure 15. Value abstraction on updates of variable x.

is changed, the probe exports the new value of x to the event recognizer. Otherwise, the probe does not. The situation is similar for y. All snapshots updating z and w, however, are exported because z and w appear in the expression $w > 2 * z + 3$ which is not a simple expression. Figure 15 illustrates value abstraction on variable x by highlighting the snapshots that are exported during a sample execution of the target pogram. The option of using value abstraction can be specified by including the `valueabstract` flag in Java-PEDL.

7. Example

We have conducted several case studies. In [11], we have monitored the emulator of a distributed controller for a large numbers of mobile agents, called Micro Air Vehicles (MAVs), in order to check whether the MAVs are forming a hexagonal pattern as required. In [3], we have detected violations of properties such as loop invariants in Ad-hoc On Demand Vector routing protocol implemented using the NS2 simulator [8]. The NS2 simulator serves as the instrumented target program and generates a trace for the event recognizer through a hand-written filter. Kim [14] monitored a simulation of an inverted pendulum (IP) specified in a hybrid specification language Charon [1] and checked whether the pendulum reaches the goal position with a rod standing upright. Furthermore, in [15] we check whether transitions among controllers occur correctly in a real IP program implemented in C/Java and hardware. Table 1 shows quantitative information on those case studies. Execution time overhead is not available for the MAV and IP case studies because we cannot measure the execution overhead accurately for these programs. These programs are periodic tasks that execute on

Table 1. Quantitive information on performed case studies.

	MAV	AODV	IP (Charon simulator)	IP
Program size (Kbyte)	41	230	980	48
Spec. size (Kbyte)	1.5	56	3	3
Size overhead	2.5%	0.1%	0.2%	0.1%
Exec overhead	N/A	N/A	1%	N/A

timer interrupts. The sleep intervals between executions mask out overheads. However, in the case of IP, which is a real-time application sensitive to the timely completion of each task invocation, the instrumentation did not lead to any violation of task deadlines. For AODV, we analyze an execution trace offline; so execution overhead is not meaningful in this case.

To help the reader understand Java-MaC, however, we describe a small, but illustrative example for Java-MaC in this paper. Consider a web-site that periodically probes some remote servers for stock quotes; the server is chosen from a list of possible servers that may provide this information, based on the web traffic at that time. On obtaining the quotes, the web-site processes the new information to compute some statistics. If it fails to obtain the quotes (due to excessive Internet traffic or the failure of the servers it accesses), it reuses old information in its processing. For such a client program, one may be interested in checking the following correctness properties:

Real-time requirement: The client is periodic; that is, it tries to query a new server every, say, 1000 msecs.

Fault tolerance requirement: Old data is used only when either the client fails to connect to some server after a sufficient number (say 3) retries or it fails to get a response from the server for the query asked after trying for, say, 4 times.

A MEDL script describing these requirements is given in figure 16. The requirements for the client can be defined provided the trace contains a signal for the beginning of the computation (`startPgm`), an event for when a fresh period of 1000 ms has started (`periodStart`), a signal when the client fails to connect to a server (`conFail`), a signal when the client resends the query (`queryResend`), and an event denoting when the client uses old information (`oldDataUSed`). Using these events, we can define the real-time requirement (`violatedPeriod`) and the fault tolerance requirement (`wrongFT`). The real-time requirement is violated whenever the time between successive `periodStart` events in the trace (stored in variable `periodTime`) is not between 900 and 1100 milliseconds. The fault tolerance requirement is defined in terms of the number of times the client failed to connect to some server (variable `numConFail`) and the number of times a query was resent (variable `numRetries`).

The run-time checker receives events `startPgm`, `periodStart`, `conFail`, `queryRe-send`, and `oldDataUsed` from an event recognizer at run-time. These events are defined in the PEDL specification of figure 17 based on methods and variables defined in `Client` class. The method `main(String[])` is invoked when the client program starts. The method `run()` is invoked when a new session begins. The metod `failConnection(ConnectTry)` is invoked when connection fails to be established. `retryGetData(int)` is invoked when the client retries to get response from the server. `processOldData()` is invoked when the old data is used instead of new data.

8. Related work

There has been two research directions for the formal analysis of program implementations. The first one is to monitor and analyze the behavior of target programs at run-time. This

```
ReqSpec StockClient
  //Imported event declaration
  import event startPgm, periodStart, conFail,
               queryResend, oldDataUsed;

  // Auxiliary variable declaration
  var long periodTime;
  var long lastPeriodStart;
  var int numRetries;
  var int numConFail;

  // Requirement definition
  alarm violatedPeriod = end((periodTime' >= 900)
                         && (periodTime' <= 1100));
  alarm wrongFT = oldDataUsed when (
              (numRetries'< 4) || (numConFail'< 3));

  // Auxiliary variable update rules
  startPgm   ->{periodTime' = 1000;
                lastPeriodStart'=
                time(startPgm) - 1000;
                numRetries' = 0;
                numConFail' = 0;}
  periodStart->{periodTime' =
                time(periodStart) - lastPeriodStart;
                lastPeriodStart'= time(periodStart);
                numRetries' = 0;
                numConFail' = 0;}
  queryResend->{numRetries' = numRetries + 1; }
  conFail ->   {numConFail' = numConFail + 1; }
End
```

Figure 16. MEDL specification for financial client example.

approach provides limited coverage because all the execution paths are not covered. This approach, however, scales up and can be a practical solution.

JASS (Java with ASSertion) [2] is a precompiler that supports boolean assertions for Java. Jass takes Java source code and inserts pre/post conditions for methods and invariants for classes in a special comments. The Java Run-time Timing constraint Monitor (JRTM) [20] aims to detect violation of timing properties in Java programs. JRTM uses Real-Time Logic (RTL) [13] as a requirement specification language. A Java program should be manually instrumented to put a probe in the place where a primitive event happens. Java Event Monitor (JEM) [17] is an event-mediator like the CORBA event channel. JEM receives predefined primitive events from event suppliers and detects composite events written in a Java Event Specification Language [18] based on these primitive events. Temporal Rover [7] monitors

```
MonScr StockClient
  // Exported event declaration
  export event startPgm, periodStart, conFail,
               queryResend, oldDataUsed;

  // Monitored methods declaration
  monmeth void Client.main(String[]);
  monmeth void Client.run();
  monmeth void  Client.failConnection(ConnectTry);
  monmeth Object Client.retryGetData(int);
  monmeth Object Client.processOldData();

  // Event definition
  event startPgm = startM(Client.main(String[] ));
  event periodStart = startM(Client.run());
  event conFail =
        startM(Client.failConnection(ConnectTry));
  event queryResend =
        startM(Client.retryGetData(int));
  event oldDataUsed =
        startM(Client.processOldData());
end
```

Figure 17. PEDL specification for financial client example.

Java/C++ programs to check whether LTL requirement specification is violated. Probes are inserted into source code manually. JavaPathExplorer [12] is a tool that is very close in spirit to the MaC tool. It also checks the correctness of Java programs with respect to LTL specifications. The correctness checking is accomplished via the rewrite engine Maude. Another approach taken is one where the events are first cpatured and then the trace is analyzed. Kortenkamp et al. [16] have a tool that allows one to capture the trace of a C/C++ program and then analyze it later with respect to formal correctness requirements. They also provide utilities that can help "visualize" the trace.

The second approach is to extract abstract models from programs written in conventional programming language such as Java. Then, extracted models are verified using model checkers. One significant advantage of this approach is that all possible execution paths of the program can be covered. This approach, however, may not scale up due to the complexity of program abstraction and state explosion problem. Bandera [6] generates finite state models in the input language of verification tool such as Spin from Java programs. These models are verified using existing model checking tools. Java Path Finder [22] extracts a finite state model from Java bytecode and applies model checking to this model against properties written in Java statements. Verisoft [10] is designed to detect a coordination problem such as deadlock and assertion violation. Verisoft generates state space systematically from C/C++ source code as long as time and space are allowed.

9. Conclusion and future work

This paper describes the Monitoring and Checking (MaC) architecture and its prototype implementation Java-MaC. The MaC architecture is a step towards bridging the gap between verification of system design specifications and validation of system implementations. The former is desirable but yet impractical for large systems, while the latter is necessary but informal and error-prone. The MaC architecture supports a light-weight formal methodology for confidence of the correct target program execution based on formal requirement specifications. In addition, the run-time analysis of the architecture produces accurate results because of the automatic processes including instrumentation, monitoring, and checking. Also, the separation between monitoring program-dependent low-level behavior from checking high-level behavior increases the flexibility of the architecture.

We have applied Java-MaC successfully to several examples including a network protocol and a micro air vehicle simulator. We are investigating application domains where we can fully exploit the features of Java-MaC effectively. We are also developing a methodology for applying the MaC architecture to support various target platforms, and extending the capability of MaC to support the steering of a target program to a safe state when a violation is detected.

Notes

1. Variables can be thought of as partial functions over time
2. Notice, that the definition of $\mathcal{D}_M^t$ refers to the definition of $\models$, and vice versa. However, the definitions are well-defined.
3. Symbolic names of local variables can be obtained from a classfile when the class is compiled with -g flag.

References

1. R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee, “Modular specifications of hybrid systems in Charon,” in *Hybrid Systems: Computation and Control*, 2000, pp. 6–19.
2. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, “Jass-Java with assertions,” in *First Workshop on Runtime Verification*, Vol. 55, No. 2, 2001.
3. K. Bhargavan, C.A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan, “Verisim: Formal analysis of network simulations,” *IEEE Transaction on Software Engineering*, Vol. 28, No. 2, pp. 129–145, 2001.
4. J. Cheng and C. Jones, “On the usability of logics which handle partial functions,” in C. Morgan and J. Woodstock (Eds.), *Proceedings of Third Refinement Workshop*. Springer-Verlag, 1991.
5. E.M. Clarke and J.M. Wing, “Formal methods: State of the art and future directions,” *ACM Computing Surveys*, Vol. 28, No. 4, pp. 626–643, 1996.
6. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsareanu, Robby, and H. Zheng, “Bandera: Extracting finite-state models from java source code,” in *Proceedings of the 22nd Int. Conf. on Software Engineering*, 2000.
7. D. Drusinsky, “The temporal rover and the ATG rover,” in *Proceedings of 7th International SPIN Workshop*, LNCS 1885, 2000, pp. 323–329.
8. K. Fall and K. Varadhan, ns Notes and Documentation, The VINT Project, 2000.
9. W.F. Farmer, “A partial functions version of Church’s simple theory of types,” *Journal of Symbolic Logic*, pp. 1269–1291, 1990.

10. P. Godefroid, “VeriSoft: A tool for the automatic analysis of concurrent reactive software,” in *Proceedings of the 9th Conf. on Computer Aided Verification*, Haifa, 1997.
11. D. Gordon, W. Spears, O. Sokolsky, and I. Lee, “Distributed spatial control and global monitoring of mobile agents,” in *Proceedings of the IEEE International Conference on Information, Intelligence, and Systems*, 1999.
12. K. Havelund and G. Rosu, “Monitoring Java programs with JavaPathExplorer,” in *Proceedigns of the Workshop on Runtime Verification*, Vol. 55 of Electronic Notes in Theoretical Computer Science. Elsevier Publishing, 2001.
13. F. Jahanian and A. Goyal, “A formalism for monitoring real-time constraints at run-time,” in *20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)*, 1990, pp. 148–55.
14. M. Kim, “Information extraction for run-time formal analysis,” Ph.D. thesis, CIS Dept. University of Pennsylvania, 2001.
15. M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky, “Monitoring, checking, and steering of real-time systems,” in *2nd Workshop on Run-Time Verification*, 2002.
16. D. Kortenkamp, T. Milam, R. Simmons, and J.L. Fernandez, “Collecting and analyzing data from distributed control programs,” in *Proceedigns of the Workshop on Runtime Verification*, Vol. 55 of Electronic Notes in Theoretical Computer Science. Elsevier Publishing, 2001.
17. G. Liu and A.K. Mok, “Implementation of JEM—A Java composite event package,” in *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, 1999.
18. G. Liu, A.K. Mok, and P.C. Konana, “A unified approach for specifying timing constraints and composite events in active real-time database systems,” in *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 1998.
19. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
20. A.K. Mok and G. Liu, “Early detection of timing constraint violation at runtime,” in *Proceedings of IEEE Real-Time Systems Symposium*, Dec. 1997.
21. D.L. Parnas, “Predicate logic for software engineering,” *IEEE Transactions on Software Engineering*, Vol. 19, No. 9, pp. 856–861, 1993.
22. W. Visser, K. Havelund, G. Brat, and S. Park, “Model checking programs,” in *Int. Conf. on Automated Software Engineering*, 2000.
23. M. Viswanathan, “Foundations for the run-time analysis of software systems,” Ph.D. thesis, CIS Dept. University of Pennsylvania, 2000.