Java-MaC: a Run-time Assurance Tool for Java Programs

M. Kim, S. Kannan, I. Lee, O. Sokolsky [1]

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, US

M. Viswanathan [2]

DIMACS & Telcordia Technologies
Piscataway, US

Abstract

We describe Java-MaC, a prototype implementation of the Monitoring and Checking (MaC) architecture for Java programs. The MaC architecture provides assurance about the correct execution of target programs at run-time. Monitoring and checking is performed based on a formal specification of system requirements. MaC bridges the gap between formal verification, which ensures the correctness of a design rather than an implementation, and testing, which only partially validates an implementation. Java-MaC provides a lightweight formal method solution as a viable complement to the current heavyweight formal methods. An important aspect of the architecture is the clear separation between monitoring implementation-dependent low-level behaviors and checking high-level behaviors against a formal requirements specification. Another salient feature is automatic instrumentation of executable codes. The paper presents an overview of the MaC architecture and a prototype implementation Java-MaC.

1 Introduction

In the past two decades, much research has concentrated on the methods for analysis and validation of software systems as such systems have been deployed in safety critical areas including avionics and automobiles. Many successful industrial case studies have been conducted in the area of formal verification [4]. Complete formal verification, however, has not yet become

[1] Email: {moonjoo,kannan,lee,sokolsky}@saul.cis.upenn.edu
[2] Email: maheshv@dimacs.rutgers.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

a prevalent analysis method. Reasons for this are twofold. First, complete verification of real-life systems remains infeasible. The growth of software size and complexity seems to exceed advances in verification technology. Second, verification results apply not to system implementations, but to formal models of these systems. That is, even if a design has been formally verified, it still does not ensure the correctness of a particular implementation of the design. This is because an implementation often is much more detailed, and also may not strictly follow the formal design. So, there are possibilities for introduction of errors into an implementation of the design that has been verified. One way that people have traditionally tried to overcome this gap between design and implementation has been to test an implementation on a pre-determined set of input sequences. This approach, however, fails to provide guarantees about the correctness of the implementation on all possible input sequences.

Consequently, when a system is running, it is hard to guarantee whether or not the current execution of the system is correct using the two traditional methods. Therefore, the approach of continuously monitoring a running system with respect to a formal requirement specification 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; just reporting a system crash is not helpful. However, run-time monitoring helps users detect and correct errors. First, subtle errors are hard to detect without thorough run-time monitoring and checking [16]. Second, errors may not cause disastrous system failure immediately. Run-time monitoring and checking can find such errors quickly and help users take a recovery action before critical failure happens.

In this paper, we describe the Monitoring and Checking (MaC) architecture whose aim is to provide assurance that the target program is running correctly with respect to a formal requirement specification. Use of formal requirement specifications in run-time monitoring is the salient aspect of the MaC architecture. The MaC architecture is a general architecture not limited to any specific programming language. To demonstrate the effectiveness of the MaC architecture, however, we have implemented a MaC prototype for Java programs called *Java-MaC*. Java-MaC instruments Java executable codes (bytecodes) automatically. This automatic instrumentation, along with automatic generation of the run-time components of Java-MaC, enables easy deployment of Java-MaC.

The paper is organized as follows. Section 2 presents an overview of the MaC architecture. Section 3 briefly presents the languages for requirement specifications. Section 4 discusses issues on how to extract information from the execution of a Java program. Section 5 describes the Java-MaC implementation. Section 6 illustrates a stock client example. Section 7 presents related work. Finally, section 8 summarizes and concludes the paper. More complete treatment of Java-MaC is given in [10].

2 Overview of the MaC Architecture

The overall structure of the architecture is shown in Fig 1. The architecture includes two main phases: *static phase* and *run-time phase*. From a target program and a formal requirement specification, the static phase (before a target program runs) automatically generates run-time components including a *filter*, an *event recognizer*, and a *run-time checker*. In the run-time phase (during the execution of a target program), information of the target program execution is collected and checked against given formal requirement specification.

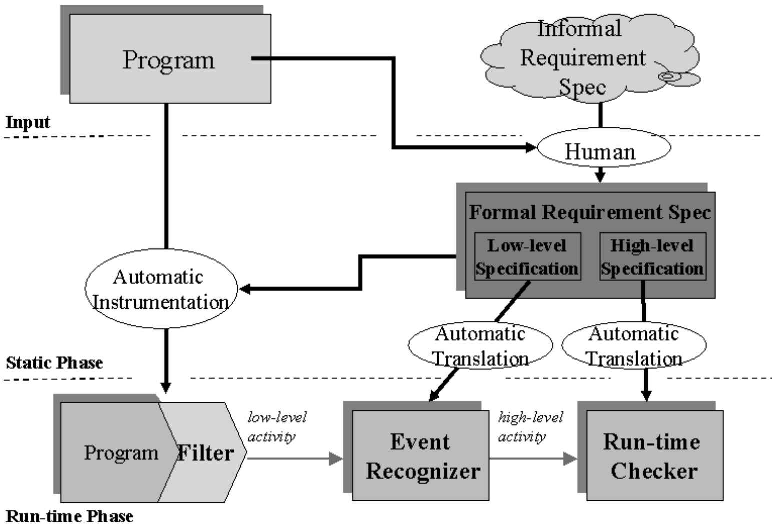

Fig. 1. Overview of the MaC architecture

2.1 Static phase

A major task during the static phase is to provide a mapping between high-level events used in the high-level requirement specification and low-level state information extracted from the instrumented target program during execution. These are related explicitly by means of a low-level specification. The low-level specification describes how events at the high-level requirement are defined in terms of monitored states of a target program. 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 target program, on the other hand, stores train's position with respect to the crossing in a variable `train_position`. The low-level specification in this case can define the event `train_in_crossing` as `train_position < 800`.

Another major task during the static phase is to generate run-time components. A filter is generated from the low-level specification and inserted into the target program automatically. An event recognizer is generated 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 the requirement 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. 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 high-level requirement 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 requirement specifications. The language for low-level requirement specification is called Primitive Event Definition Language (PEDL). PEDL specifications are used to define what information is sent from the filter to the event recognizer, and how it is transformed into events used in high-level requirements specification by the event recognizer. High-level requirement specifications are written in Meta Event Definition Language (MEDL). The primary reason for having two separate languages in the MaC architecture is to separate implementation-specific details of monitoring from high-level requirements checking. This separation ensures that the architecture is portable to different implementation languages and specification formalisms, while providing a clean interface to the designer of monitors.

Before presenting the two languages, PEDL and MEDL, we discuss some key features of these languages. In Sec 3.1, we illustrate the distinction between *events* and *conditions*.

3.1 Events and Conditions

As described in Section 2.2, whenever an "interesting" state change occurs in the target system, a filter sends a notification to an event recognizer. Based on updates from the filter, a monitor consisting of an event recognizer and a run-time checker matches the trace of the current execution against the requirements. In order to do this, we distinguish between two kinds of state information underlying the notifications.

Events occur instantaneously during the system execution, whereas *conditions* 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` does not change its value from 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.

We have two attributes `time` and `value`, defined for events. 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. `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 perfectly 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.

We assume a countable set $\mathcal{C} = \{c_1, c_2, \ldots\}$ of primitive conditions. For example, in PEDL for Java (see Sec 4.2), these primitive conditions will be Java boolean expressions built from the monitored variables. In MEDL (see Sec 3.3), 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. Primitive events in PEDL for Java (see Sec 4.2) correspond to updates of monitored variables and calls/returns of monitored methods. The primitive events in MEDL are those that are reported by the event recognizer. Table 3.1 shows the syntax of conditions (C) and events (E).

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

Table 1
The syntax of conditions and events

During any execution, variables routinely become undefined when they are out of scope. We choose to use a three-valued logic, where the third value is taken to represent undefined (Λ). We interpret conditions over three values, $true$, $false$, and Λ. The predicate $\texttt{defined}(c)$ is true whenever the condition c has a well-defined value, namely, $true$ or $false$. Negation ($!c$), disjunction ($c_1 \Vert c_2$), 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 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 events, conjunction ($e_1 \&\& e_2$) and disjunction ($e_1 || e_2$) are 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 $true$ ($\mathtt{start}(c)$), and the instant when the condition becomes $false$ ($\mathtt{end}(c)$). Notice that the event corresponding to the instant when the condition becomes Λ can be described as $\mathtt{end}(\mathtt{defined}(c))$. 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 MaC reasons about temporal behavior and data behavior of the target program execution using events and conditions; events are abstract representation of time and conditions are abstract representation of data. For semantics of events and conditions, see [9,11].

3.2 Primitive Event Definition Language (PEDL)

PEDL is the language for writing low-level requirement specifications. PEDL is based on events and conditions. 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. All 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 Sec 4.2.

3.3 Meta Event Definition Language (MEDL)

The safety requirements (invariants) are written in MEDL. MEDL is based on events and conditions as PEDL is. 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 Fig 2.

First, a list of events and conditions to be imported from an event recognizer is declared. Events and conditions are defined using imported events, imported conditions, and auxiliary variables, whose role is explained later in this section. These events and conditions are then used to define safety properties and alarms. The correctness of the system is described in terms of safety properties and alarms. Safety properties are conditions that must *always* be

```
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
```

Fig. 2. Structure of MEDL

true during the execution. Alarms, on the other hand, are events that must never be raised. [3] 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.

The language described in Sec 3.1 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. [4] Updates of auxiliary variables are triggered by events.

4 Information Extraction from Java Programs

This section describes methodology of extracting information from the execution of a Java program. First, design issues of Java-MaC relevant to the object orientation and the multi-thread of the Java programming language are described. Then, we describe PEDL for Java, in which low-level specifications containing Java specific descriptions are written. Then, we discuss how to monitor objects and instrument Java programs according to PEDL for Java specifications.

[3] All safety properties [14] can be described in this way.

[4] MEDL describes an automaton extended with auxiliary variables.

4.1 Design Issues of Java-MaC

4.1.1 Object Aliasing

A Java program forms a complex object graph. Java handles an object via a *reference* pointing to the object. An object contains variables of primitive types and references to other objects. It is non-trivial to specify and monitor a variable in a complex object graph.

Let us see how to specify and monitor the variable `x` pointed by an arrow in Fig 3. First, we specify `x`'s location (parent object) in the object graph such as `a.b2` to distinguish from `x` in another object such as `a.b1`. Second, we need to monitor all references to the parent object of `x` such as `a.b1.b'` and `a.b2`.

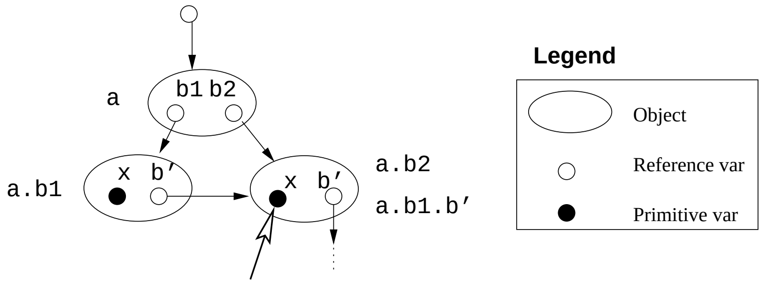

Fig. 3. An object graph

A monitored variable can be updated through several different references pointing to the parent object of the variable. Thus, references which possibly point to the parent object need to be tested at run-time to see whether they are actually pointing to the parent object. This testing, however, may not be feasible. A reference to the parent object may not be visible to locations where other references of the same type are updated due to Java scoping rules. Suppose that `b2` is declared as `private` in the class `A`. Then, we cannot test whether `a.b1` is equal to `a.b2` outside of the class `A`.

4.1.2 Preemptions in Multi-threaded Programs

The Java programming language is a multi-threaded programming language. Due to preemptions in thread scheduling, a filter may report variable updates of concurrent threads differently from what really happens in the target program. Fig 4 shows such an example. `ldc 10` loads a constant 10 onto the top of an operand stack. `putfield x` updates `x` with the top element in the operand stack. `send_update()` reports monitored variable update to an event recognizer.

The update of `x` should be reported earlier than that of `y`. However, the update of `y` is reported earlier than that of `x` because thread 1 is preempted just before reporting the update of `x`. Furthermore, if thread 2 had a instruction `putfield x` instead of `putfield y`, thread 2 would overwrite `x` as 20; a filter would miss a snapshot of `x` as 10.

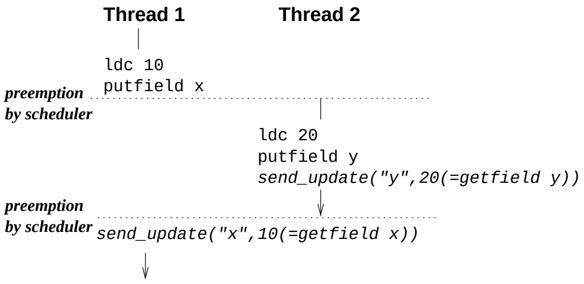

Fig. 4. Incorrect ordering of update reports

4.2 PEDL for Java

PEDL is closely related to the target programming language because events are defined using programming entities such as variables and methods. In this subsection, we will simply use PEDL as PEDL for Java. PEDL is designed for automatic instrumentation, guarantee of no harmful side effects, [5] and fast recognition of events. Thus, PEDL does not allow recursive expressions and quantifiers. We believe that this lack of expressive power is moderate so that the user still can monitor and check interesting properties. The overall structure of a PEDL specification is shown in Fig 5.

```
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
```

Fig. 5. Structure of PEDL

The *export section* declares a list of events and conditions to export to an event recognizer. The *overhead reduction section* sets flags to reduce monitoring overhead (see [10] for details).

[5] Java-MaC has side effects on resource consumption such as memory and CPU time. However, Java-MaC guarantees that program variables are not modified by Java-MaC. In addition, Java-MaC does not change the control flow of the program unless the program has synchronization errors.

4.2.1 Declared Monitored Variables and Methods

PEDL does not monitor objects directly but monitors primitive variables for reducing monitoring overhead. Note that an object possibly contains references to other objects and forms a graph. The first overhead of monitoring an object is that we must keep watching whether any node in the object graph is being changed. In addition, when we detect that the object has changed (i.e., some node in the object graph has changed), the object graph should be delivered to the event recognizer. This can result in too much overhead. Henceforth, whenever we say monitoring an object, we mean monitoring primitive variables of the object.

PEDL declares *execution points* to be monitored. PEDL uses beginnings/endings of methods as monitored execution points rather than source code line number. This is because source code is not usually available outside of the developer of a target program. Furthermore, a line number does not have inherent meaning in the target program (ex. insertion of a dummy line changes line numbers).

4.2.2 Defining Events and Conditions

Basic building blocks of events and conditions in PEDL specification are primitive variables and methods declared as monitored entities.

Defining Conditions

Primitive conditions in PEDL are constructed from boolean-valued expressions over the monitored variables. An example of such condition is `condition TooFast = Train.calculatePosition().trainSpeed>100`. In addition to these constructed boolean expressions, we have the primitive condition `InM(f)`. This condition is true as long as the execution is currently within the 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/returns of monitored methods. The event `update(x)` is triggered when variable x is assigned a value. Events `startM(f)` (`endM(f)`) are triggered when control enters method `f` (respectively, returns from `f`). For example, `event OpenGate = startM(Control.open())` defines an event meaning a controller starts opening a gate.

All operations on events in Table 3.1 can be used to construct more complex events from these primitive events. PEDL has two attributes defined for events, `time` and `value`. `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 occured. `value(e,i)` gives the ith value in the tuple of values associated with `e`, provided `e` occurs.

4.3 Monitoring Objects

Java-MaC creates a globally accessible table containing addresss of monitored objects and monitored object names, called *address table*. The address table should be updated at run-time following reference changes. The cost of updating the address table can be very expensive, because one reference change can cause huge substitution of all descendent nodes already in the address table. Suppose that an object `a` has two references `b1` and `b2` of the same type as its member variables as in Fig 3. Java-MaC can distinguish an object referred by `a.b1` and another object referred by `a.b2` by comparing the addresses at run-time. Since an object is located at a unique address in the heap, comparing two objects' addresses allows us to distinguish one object from another. Suppose that we want to monitor `a.b1` where `a.b1` is located at heap address 8200 and `a.b2` is located at heap address 8300. Let us suppose the address table contains (8300, `a.b2`). [6] At run-time, we can check whether a reference `b` has an address 8300 or not. If `b`'s address is 8300, `b` points to the same monitored object `a.b2`.

The address table should be updated whenever a monitored reference is updated. However, as we have seen in Sec 4.1.1, this can cause daunting overhead. Java-MaC puts restriction in order to avoid this overhead: a reference should not change. We believe that this restriction does not severely limit Java-MaC. In fact, several case studies including validation of network routing protocol [3] and mobile physical agents simulation [6] have been successfully conducted with this limitation.

4.4 Instrumentation Process

Java-MaC monitors *global primitive variables* declared as members of a class, *local primitive variables* declared inside methods, and *beginnings/endings of methods*. The Java-MaC instrumentor detects instructions which update monitored variables or instructions located at the beginnings/endings of methods. Global primitive variables are updated by `putstatic` for a static variable or `putfield` for a member variable. Local primitive variables are updated by `<T>store`, `<T>store_<n>` and `iinc`. The instrumentor inspects instruction codes and parameters and find candidate update instructions for monitored variables.

Once Java-MaC instrumentor recognizes a candidate update instruction for a monitored variable, the instrumentor inserts `monitorenter` and `monitorexit` for making update of a variable and report of update an atomic session (see Sec 5.2.1). Also, the instrumentor inserts a probe invoking `sendObjMethod(Object parentAddress, <T> value, String varName)`. [7] For execution poionts,

[6] In general, an address can match more than one monitored object's name due to aliasing. We will assume, however, that an address can match only one monitored object's name. The validity of this assumption follows from the fact that we do not allow reference assignment.

[7] `parentAddress` is an address of an object whose member field `varName` is monitored.

the instrumentor inserts probes at the starting point of a method (beginning of a method definition) and at the ending points of a method (locations where `return` instructions exist).

5 The MaC Prototype for Java

This section describes the MaC prototype for Java programs, called *Java-MaC*. The overall structure is depicted in Fig 6. Sec 5.1 describes Java-MaC components of static phase. Sec 5.2 describes run-time components of Java-MaC.

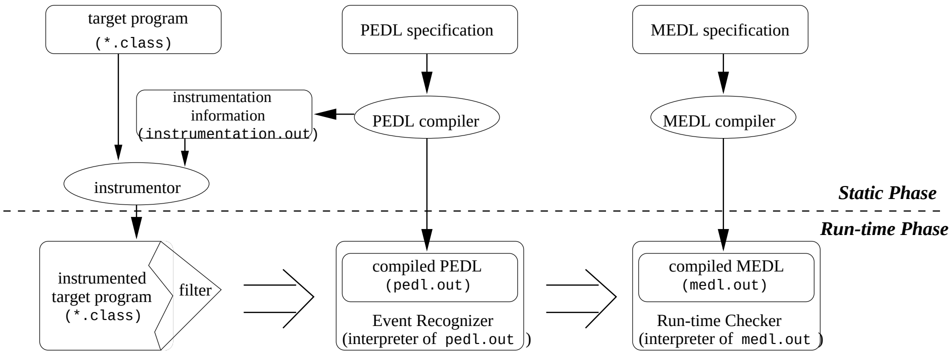

Fig. 6. Java-MaC

5.1 Static Phase

Java-MaC has three static phase components: an *instrumentor*, a *PEDL compiler*, and a *MEDL compiler*. A Java-MaC instrumentor takes a Java bytecode (`*.class`) and instrumentation information (`instrumentation.out`) containing a list of monitored variables/methods and monitoring flags generated from a PEDL specification. Based on these two inputs, the Java-MaC instrumentor inserts a filter into the target bytecode. A PEDL compiler compiles a PEDL specification into an abstract syntax tree (`pedl.out`) which is evaluated by an event recognizer at run-time. At the same time, a PEDL compiler generates instrumentation information (`instrumentation.out`) which is used by the instrumentor. Similarly, a MEDL compiler compiles a MEDL specification into an abstract syntax tree (`medl.out`) which is evaluated by a run-time checker at run-time.

`sendObjMethod()` checks whether a variable this probe monitors is actually the monitored variable (or beginning/ending of a method) by checking if `parentAddress` matches the address of a monitored object in the address table (see Sec 4.3). If the variable is a monitored variable, `sendObjMethod()` sends it to the event recognizer. Otherwise, not.

5.2 Run-time Phase

5.2.1 Filter

A filter extracts snapshots from the target program execution and sends these snapshots to the event recognizer. A filter consists of the following three parts: *a communicationo channel, probes*, and *a filter thread*. A target program is not designed to communicate with an event recognizer originally. A communication channel from the target program to the event recognizer is created by a filter. Probes extract the new value of a monitored variable and sends the value (or beginning/ending signal of a monitored method) to the event recognizer through the communication channel. A filter thread flushes the content of the communication buffer to the event recognizer.

A filter makes update of variable and report as an atomic action to prevent incorrect ordering of reports and overwriting (see Sec 4.1.2 for problem description). An atomic session is implemented using a global lock. Thread t acquires the lock right before executing an update instruction. After finishing the update and report, t releases the lock. When preemption happens while t is reporting, no other thread t' can make a report until t finishes.

5.2.2 Event Recognizer

Whenever an event recognizer receives snapshots from a filter, the event recognizer evaluates events and conditions by traversing the abstract syntax tree in `pedl.out`. PEDL expressions are evaluated in linear time in terms of the size of expressions since PEDL does not allow recursion. If the event recognizer detects events, the event recognizer sends the events to the run-time checker. Similarly, the event recognizer sends conditions changed to $true$, $false$, or Λ.

5.2.3 Run-time Checker

A 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 also, because, like PEDL, MEDL does not allow recursive expressions. If the run-time checker detects a violation defined by `alarm` or `property`, the run-time checker raises a signal.

5.2.4 Connection of Run-time Components

Connections among Java-MaC run-time components are established before a target application executes. The choice of communication medium is important because a communication medium affects the correctness of checking. If a communication medium does not guarantee delivery of messages in order, the correctness of checking may not be guaranteed either. In addition, a communication medium affects the performance of monitoring. If a communication is established through TCP socket, it may pose relatively large overhead compared to one using shared memory. Furthermore, the communication medium may have to satisfy certain constraints (ex. security).

Java-MaC provides three different communication mechanisms among the run-time components: TCP socket communication, communication through a FIFO file, and communication channel implemented by a user using `InputStream` and `OutputStream` provided by Java-MaC API.

5.3 Monitoring Overhead

Monitoring activity causes unavoidable overhead to the target system execution unless specialized hardware is utilized. The overhead depends on several factors including frequency of taking snapshots, nature of communication medium, and evaluation speed of properties. The overhead of Java-MaC is less than 10% when the frequency of taking snapshot is once per 10^5 bytecode execution. [8]

6 Example: Financial Client

We describe a small, but illustrative example for Java-MaC in this section. [9] 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 the web-site 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, every few (say 1000 ms) seconds it tries to query a new server.

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

A MEDL script describing these requirements is given in Fig 7. 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`)

[8] Communication between the Java-MaC run-time components uses TCP socket in this measurement. More details on overhead analysis and overhead reduction techniques, see [10].

[9] For Java-MaC case studies, see [6,10,3].

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

  // Auxiliary variable declartion
  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
```

Fig. 7. MEDL specification for Financial Client example

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`).

A run-time checker receives events `startPgm`, `periodStart`, `conFail`, `queryResend`, and `oldDataUsed` from an event recognizer at run-time. These events are defined in the PEDL specification of Fig 8 based on methods and variable defined in `Client` class. A method `main(String[])` is invoked when the client program starts. `run()` is invoked when a new session begins. `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.

7 Related Work

There are 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 approach provides limited coverage because all the execution paths are not covered. However, this approach scales up well

```
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
```

Fig. 8. PEDL specification for Financial Client example

and can be a practical solution. A Lightweight Architecture for MOnitoring (ALAMO) [8] instruments C source code automatically according to the configuration written by a user. The configuration language (similar to PEDL) declares what activities are to be recognized as events. ALAMO, however, does not provide a high-level formal specification language such as MEDL. 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) [15] aims to detect violation of timing properties in Java programs. JRTM uses Real-Time Logic (RTL) [7] 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) [13] 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 [12] based on these primitive events. Time Rover [1] monitors Java/C++ programs to check whether LTL requirement specification is violated or not. Probes are inserted into source code manually.

The second approach is to extract models from programs written in conventional programming language such as Java. Then, extracted models are verified using model checkers. A strong point of this approach is that all possible execution paths of the program can be covered. However, this approach may not scale up well due to complexity of program abstraction and state explosion problem. Bandera [5] 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 [17] extracts a finite state model from Java bytecode and applies model checking to this model against properties written in Java statements.

8 Conclusion

This paper describes the Monitoring and Checking (MaC) architecture and its prototype implementation Java-MaC. Monitoring and checking is performed based on a formal specification of system requirements. 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 assuring the correctness of the current execution of a target program based on formal requirement specifications. The MaC architecture uses layerd approach. The architecture separates monitoring program-dependent low-level behavior from checking high-level behavior. This separation makes the MaC architecture an extendable open architecture applicable to broad range of target platforms. Finally, the automatic generation of the run-time components in Java-MaC makes deployment of Java-MaC easy and practical. 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. At the same time, we are investigating the methodology of applying the MaC architecture to support various target platforms other than Java.

References

[1] Time rover home page, 1997. `http://www.time-rover.com/`.

[2] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass- Java with Assertions. *First Workshop on Runtime Verification*, July 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*, 2001. To be published.

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

[5] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proc. of the 22nd Int. Conf. on Software Engineering*, June 2000.

[6] D. Gordon, W. Spears, O. Sokolsky, and I. Lee. Distributed spatial control and global monitoring of mobile agents. In *IEEE Int. Conf. on Information, Intelligence, and Systems*, Nov 1999.

[7] F. Jahanian and A. Goyal. A formalism for monitoring real-time constraints at run-time. *20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)*, pages 148–55, 1990.

[8] Clinton Jeffery, Wenyi Zhou, Kevin Templer, and Michael Brazell. A lightweight architecture for program execution monitoring. *Program Analysis for Software Tools and Engineergin*, July 1998.

[9] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *European Conf. on Real-Time Systems*, 1999.

[10] Moonjoo Kim. *Information Extraction for Run-time Formal Analysis.* PhD thesis, CIS Dept. Univ. of Pennsylvania, 2001. In preparation.

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

[12] 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 *RTAS*, June 1998.

[13] Guangtian Liu and Aloysius K. Mok. Implementation of jem - a java composite event package. In *RTAS*, 1999.

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

[15] Aloysius K. Mok and Guangtian Liu. Early detection of timing constraint violation at runtime. In *RTSS*, Dec 1997.

[16] President's Information Technology Advisory Commitee. *Information Technology Research: Investing in Our Future*, 1999. `http://www.ccic.gov/`.

[17] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Int. Conf. on Automated Software Engineering*, September 2000.