Electronic Notes in Theoretical Computer Science 70 No. 4 (2002)
URL: `http://www.elsevier.nl/locate/entcs/volume70.html` 15 pages

Computational Analysis of Run-time Monitoring
- *Fundamentals of Java-MaC* *

Moonjoo Kim [1]

SECUi.COM R&D Center, Seoul Korea

Sampath Kannan, Insup Lee, Oleg Sokolsky [2]

University of Pennsylvania, Philadelphia US

Mahesh Viswanathan [3]

University of Illinois at Urbana-Champaign, Urbana US

Abstract

A run-time monitor shares computational resources, such as memory and CPU time, with the target program. Furthermore, heavy computation performed by a monitor for checking target program's execution with respect to requirement properties can be a bottleneck to the target program's execution. Therefore, computational characteristics of run-time monitoring cause a significant impact on the target program's execution.

We investigate computational issues on run-time monitoring. The first issue is the power of run-time monitoring. In other words, we study the class of properties run-time monitoring can evaluate. The second issue is computational complexity of evaluating properties written in process algebraic language. Third, we discuss sound abstraction of the target program's execution, which does not change the result of property evaluation. This abstraction can be used as a technique to reduce monitoring overhead. Theoretical understanding obtained from these issues affects the implementation of Java-MaC, a toolset for the run-time monitoring and checking of Java programs. Finally, we demonstrate the abstraction-based overhead reduction technique implemented in Java-MaC through a case study.

⋆ This research was supported in part by ONR N00014-97-1-0505, NSF CCR-9988409, NSF CCR-0086147, NSF CISE-9703220, and ARO DAAD19-01-1-0473.

[1] Email: `moonjoo@secui.com`

[2] Email: {`kannan,lee,sokolsky`}`@saul.cis.upenn.edu`

[3] Email: `vmahesh@cs.uiuc.edu`

1 Introduction

The purpose of run-time analysis based on formal requirements is to bridge the gap left by the two traditional analysis techniques: verification and testing. On the one hand, formal verification analyzes all possible executions of the system, but the analysis is performed on the specification of the system, not its implementation. In addition, state-of-the-art verification techniques still do not scale up well to handle large systems. On the other hand, testing is applied to a system implementation, but does not guarantee that all behaviors of the implementation are explored and analyzed. In either case, run-time analysis can provide additional assurance that the current execution is correct with respect to its requirements. Run-time analysis can find violations to the requirements in time and help users to take recovery actions before critical failure happens.

While this additional assurance is an invaluable help to detect and recover errors, this benefit comes at the cost of slowed target program's execution due to monitoring overhead. Monitoring overhead consists of two factors: *information extraction overhead* and *evaluation overhead*. Information extraction overhead occurs due to the execution of probes inserted into the target system. Evaluation overhead occurs when a monitor evaluates requirement properties. Unless special hardware is attached to the target system for intercepting execution information, probes inserted into the target system share computational resources, such as memory and CPU time, of the target system to extract execution information. Evaluation overhead, however, occurs even when special hardware is utilized. Suppose that the requirement properties are large and complex. Then, the speed of evaluating the properties is slower than the execution speed of the target program, which makes the target program wait until the evaluation finishes. Probes send snapshots of the target program's execution to the (finite) snapshot buffer based on which a monitor evaluates the properties. When the snapshot buffer becomes full, probe should stop the target program's execution until the buffer has room. Some monitoring system [2] prevents the target system from waiting for the monitor by overwriting snapshots while a monitor is busy to evaluate properties. This approach, however, cannot guarantee correct evaluation and is not adequate for run-time verification. As described above, computational characteristics of run-time monitoring, especially property evaluation, cause a significant impact on target program execution.

We investigate computational issues on run-time monitoring which are not limited to specific monitoring architecture, but universal. The first issue is on the computational power of run-time monitoring. In other words, we investigate the class of properties which run-time monitoring can evaluate at run-time, i.e, in finite time. We show that this class of properties does *not* coincide with safety properties, but a strict subset of safety properties. The second issue we study is computational complexity of evaluating properties written in process algebraic language. We formulated this problem as a trace validity problem and proved that the evaluating properties written in CCS is a NP-complete problem using 3SAT problem. As far as we know, these results are new contributions to the run-time monitoring field. Finally, we develop a sound abstraction technique on target program execution so that a monitor evaluates properties less frequently but does not miss violation to the properties. This technique, called *value abstraction*, can be used to reduce monitoring overhead. We demonstrate this overhead reduction technique implemented in Java-MaC in the case study of monitoring the Sieve of Eratosthenes program.

Section 2 shows that the class of properties run-time monitoring can evaluate is a strict subset of safety properties. Section 3 proves that the problem of evaluating properties written in process algebra is NP-complete. Section 4 explains value abstraction. Section 5 briefly describes Java-MaC and the impact of previously mentioned issues on Java-MaC. Section 6 shows the experimental result of applying value abstraction to monitor the Sieve of Eratosthenes. Finally, Section 7 concludes this paper.

2 A Class of Monitorable Properties

Run-time monitoring has weaker power of evaluating requirement properties than verification has. It is because run-time monitoring observes target program execution of *finite* length while verification searches whole (possibly *infinite*) execution. Thus, it is obvious that run-time monitoring cannot evaluate liveness properties. We generally presume that the class of properties run-time monitoring can evaluate is safety properties. In this section, however, we study the class of properties run-time monitoring can evaluate more precisely. We call a property run-time monitoring can evaluate as "a monitorable property". We start discussion on monitorable properties by defining "execution" and "property" formally.

Definition 1 (Execution) *An execution of a program is an infinite sequence of program states $\sigma = s_0 s_1 ...$ where $s_i \in S$ (a set of program states), $s_0 \in S_{init}$ (a set of initial states), and $\sigma[i..j]$ is the subsequence of σ from a state s_i to a state s_j.*

The definition of an execution can apply to finite sequences by obtaining an infinite sequence from a finite one by repeating the final state of the finite sequence. This corresponds to the view that a terminating execution is the same as non-terminating execution in which after some finite time (once the program has terminated) the state remains fixed.

Definition 2 (Property) *A* property *is a set of executions. We write $\sigma \models P$ to denote that σ is in property P.*

Let us define safety property formally. Informally speaking, safety property means that bad things do not happen during the execution of a program. Consider a safety property P_{safe} that means some bad thing x does not happen. If $\sigma \not\models P_{safe}$, σ includes some bad thing which cannot be remedied. In other words, there is some prefix of σ which includes some bad thing for which no extension will satisfy P_{safe}. We use S^ω as the set of infinite sequences of states.

Definition 3 (Safety Property) *A property $P \subseteq S^\omega$ is a safety property if for every $\sigma \in S^\omega$, $\sigma \in P$ if and only if $\forall i \exists \beta \in S^\omega(\sigma[0..i]\beta \in P)$ where S is the set of program states.*

It is clear from Definition 3 that a monitorable property is a safety property. A monitor can evaluate a property based on only a finite number of execution states. A safety property, however, is *not* necessarily a monitorable property. The definition of safety property makes *no computational assumptions*; it is possible to define a property that is a safety property, but which is unlikely to be monitorable. Safety closure of the halting problem is a safety property but not a monitorable property.

Example 1 *Let $\Sigma = \{0, 1, a, b\}$. Consider a finite property $H_* = \{x \cdot a \cdot y \mid x, y \in \{0, 1\}^*$, the Turing Machine encoded by x halts on input $y\}$. We define a property $H_\omega = H_* \cdot b^\omega \cup \{0, 1\}^* \cdot a \cdot \{0, 1\}^\omega \cup \{0, 1\}^\omega$.*

The property H_ω, defined above is a safety property. In order to see this, we only need to observe that for any execution not in H_ω, there is finite prefix when this violation can be detected. Executions not in H_ω are those that are not in the "right format", or where the finite prefix before the sequence of b's is not in H_*; in both cases there is a finite prefix that provides evidence of the execution not being in the property.

However, in order to detect that an execution σ is not in H_ω, we have to check for membership in H_*. Since membership in H_* (or the Halting problem) is not decidable, it is impossible for us to design monitors that would be able to detect a violation of this property. This suggests that the class of *monitorable properties* is a strict subset of a class of safety properties (see Figure 1); they should be such that sequences not in the properties should be recognizable by a Turing Machine, after examining a finite prefix. Therefore, we can define a monitorable property as follows.[4] We use pref(σ) for $\sigma \in S^\omega$ as the set of all finite prefixes of σ.

Definition 4 (Monitorable Property) *A property $P \subseteq S^\omega$ is said to be* monitorable *if and only if P is a safety property and $S^* \setminus \mathrm{pref}(P)$ is recursively enumerable, where* $\mathrm{pref}(P) = \cup_{\sigma \in P} \mathrm{pref}(\sigma)$

3 Evaluation of Properties in Process Algebra

Requirement properties need to be described in a property specification language. The characteristics of the property specification language can affect the

[4] For more detailed discussion on monitorable language, see [8].

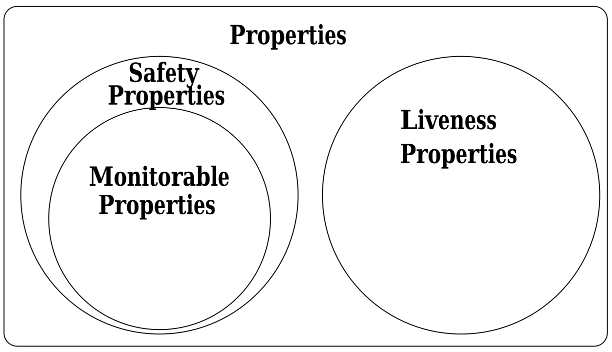


Fig. 1. Monitorable properties

computational complexity of evaluating properties. [7] shows the monitoring overhead due to non-determisim in requirement properties in their monitoring system, called Supervisor, and provides heuristics to decrease the overhead. [3] develops a linear time monitoring algorithm for safety formulae written in past time LTL. In this section, we will discuss the computational complexity of monitoring algorithm for property written in process algebra.

Run-time monitoring can be thought of as a *trace validity problem* where a trace is generated from the execution of the program. The trace validity problem is a membership checking problem of deciding whether a given trace is in the set of valid traces. For sufficiently expressive requirement specification languages such as CCS [6] or ACSR [1], this problem turns out to be NP-complete. We formulate the trace validity problem using the notation of [6].

3.1 Notations

We will denote the *ith* character in a string x by $x^{(i)}$. $\mathcal{A}$ is an set of names $a, b, c, ...$ Then, $\overline{\mathcal{A}}$ is the set of *co-names* $\overline{a}, \overline{b}, \overline{c}, ...$; $\mathcal{A}$ and $\overline{\mathcal{A}}$ are disjoint and are in bijection via $(^{-})$; we declare $\overline{\overline{a}} = a$. $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ denotes the set of labels. We also introduce a distinguished silent action $\tau \notin \mathcal{L}$. We set $Act = \mathcal{L} \cup \{\tau\}$.

Definition 5 *The set of processes is defined by*

$$P \ ::= \ Nil \mid \alpha.P \mid P \ + \ Q \mid P||Q \mid P\backslash L$$

where $L \subseteq \mathcal{L}$ *and* $\alpha \in Act$.

Definition 6 *The labeled transition relation* $\xrightarrow{\alpha}$ *between two processes is defined by the following rules. In the following rules,* $\alpha \in Act, l \in \mathcal{L}$, *and* $L \subseteq \mathcal{L}$.

$$[Prefix]\frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$[Choice]\frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \quad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

$$[Parallel]\frac{P \xrightarrow{\alpha} P'}{P||Q \xrightarrow{\alpha} P'||Q} \quad \frac{Q \xrightarrow{\alpha} Q'}{P||Q \xrightarrow{\alpha} P||Q'} \quad \frac{P \xrightarrow{l} P',\ Q \xrightarrow{\overline{l}} Q'}{P||Q \xrightarrow{\tau} P'||Q'}$$

$$[Restriction]\frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'} \text{ where } \alpha \notin L \cup \overline{L}$$

Definition 7 *Given processes P and P', and $\alpha \in \mathcal{L}$, we say that $P \overset{\alpha}{\Rightarrow} P'$ if $P(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* P'$, where $(\xrightarrow{\tau})^*$ is the transitive reflexive closure of $\xrightarrow{\tau}$.*

3.2 Trace Validity Problem

In this section, we formulate the trace validity problem using the notations in Section 3.1.

Definition 8 (Valid Trace) *A string $s \in \mathcal{L}^*$, of length n, is said to be a valid trace of a process P, if there exist processes $P_0, P_1, \ldots, P_n$, such that $P \equiv P_0$, and $P_{(i-1)} \overset{s^{(i)}}{\Rightarrow} P_i$, for all $i \in \{1, \ldots, n\}$*

Then, the Trace Validity Problem is formally defined as follows:

Input	A process P and a string $s \in \mathcal{L}^*$.
Output	Is s a valid trace of P?

Theorem 3.1 *The trace validity problem is* **NP***-complete.*

Proof.

To prove hardness, we reduce 3SAT to the trace validity problem. We are given a formula φ in conjunctive normal form with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, each with three literals. We construct a process $P(\varphi)$ and a string $s(\varphi)$ such that $s(\varphi)$ is a valid trace of $P(\varphi)$ iff the formula φ is satisfiable.

For each i, define processes, X_i, as follows,

$X_i \overset{\text{def}}{=} \tau.F_i + \tau.T_i$

$F_i \overset{\text{def}}{=} \overline{f_i}.F_i$

$T_i \overset{\text{def}}{=} \overline{t_i}.T_i$

In our reduction, these processes express a truth value assignment to the variables. If the $X_i \xrightarrow{\tau} F_i$ then it expresses the fact that under this assignment the variable x_i gets the value **false**, and if $X_i \xrightarrow{\tau} T_i$ then it means that the variable x_i gets the value **true**.

In addition to these processes, we define another process, P. The idea is that P will deadlock, when run concurrently with the processes X_i, iff the truth assignment defined (as above) by the processes X_i is not a satisfying truth assignment for the formula φ.

In order to define the process P, we assume that $C_i \equiv l_{i,1} \vee l_{i,2} \vee l_{i,3}$ for $i \in \{1, \ldots m\}$ and $j \in \{1, 2, 3\}$ in following formulas.

$$
\begin{aligned}
P &\stackrel{\text{def}}{=} Q_1 \\
Q_1 &\stackrel{\text{def}}{=} a.L_{1,1} + a.L_{1,2} + a.L_{1,3} \\
&\vdots \\
Q_i &\stackrel{\text{def}}{=} a.L_{i,1} + a.L_{i,2} + a.L_{i,3} \\
L_{i,j} &\stackrel{\text{def}}{=} \begin{cases} f_k.L'_{i,j} \text{ if } l_{i,j} \equiv \neg x_k \\ t_k.L'_{i,j} \text{ if } l_{i,j} \equiv x_k \end{cases} \\
L'_{i,j} &\stackrel{\text{def}}{=} b.Q_{i+1} \\
&\vdots \\
L'_{m,j} &\stackrel{\text{def}}{=} b.Q_1
\end{aligned}
$$

The process $P(\varphi)$ is thus $(P||X_1|| \cdots ||X_n) \setminus \{t_1, f_1, \ldots, t_n, f_n\}$. The property this process has is that, for any i, j, the transition $L_{i,j} \to L'_{i,j}$ can be taken iff the literal $l_{i,j}$ gets the truth value **true** under the truth assignment defined by the processes $X_1, \ldots, X_n$. Hence, $Q_i \to^* Q_{i+1}$ can take place iff one of the literals in the clause C_i gets the truth value **true** under the assignment described by $X_1, \ldots, X_n$. Thus it can be seen that $abab \ldots ab$ is a valid trace of $P(\varphi)$ iff φ has a satisfying assignment.

To prove completeness, we prove that Trace Validity Problem belongs to NP. We can view a process P as a labeled transition graph G_P over a set of label $\mathcal{L}$ rooted at the node n_P. For a given process P and a string $s \in \mathcal{L}^*$, we choose a path p corresponding to s from n_P be the certificate. Checking can be accomplished in polynomial time by traversing G_P from n_P following p.

□

The main reason for NP-completeness in the trace validity problem is non-determinism caused by parallel composition of processes. We should be careful to define or use a requirement property specification language so that trace validation against properties is tractable.

4 Abstraction of Program Execution

Computational characteristic of run-time monitoring is closely related to *abstract view* on the target program execution. In this section, we provide an abstract view on the target program execution. An execution of a program is a sequence of program states as defined in Definition 1. We define a state as follows.

Definition 9 (State) *A state s of a program execution is a pair of a time stamp $t_s \in \mathcal{R}$ and an environment $\rho_s \subseteq V \to \mathcal{R}$ which is a function from a set of variables V to a set of real values $\mathcal{R}$.*

A state in an execution indicates variable change(s) at the time instant corresponding to the state. ρ_{s_i}, however, does not change between time interval starting from a state s_i until the next state s_{i+1}; the information of a program remains fixed between two states s_i and s_{i+1}. If we keep track of variable updates, we can capture snapshots of s_i's.

Note that not all variables relate to requirement properties. A property p_1 may consist of expressions based on a variable v_1, but not v_2. Furthermore, not every update of v_1 changes evaluation of p_1. In other words, a monitor can abstract out states whose updated variable values do not affect evaluation of p_1. We call this abstraction *value abstraction.* Value abstraction is a function from an execution to a shorter execution consisting of less states. Let us formulate value abstraction formally. We will use S^∞ denoting $S^* \cup S^\omega$. *Value abstraction* $\gamma_{exp_{V_m}}$ abstracts out states which do not affect exp_{V_m}, a set of boolean expressions over the monitored variables V_m.

Definition 10 (Value Abstraction) *A value abstraction $\gamma_{exp_{V_m}}$ with regard to exp_{V_m}, a set of boolean expressions over monitored variables V_m is a function $\gamma_{exp_{V_m}} : S^\omega \to S^\infty$. $\gamma_{exp_{V_m}}$ is defined recursively as follows.*

$$\gamma_{exp_{V_m}}(s_i s_{i+1}\sigma') = \begin{cases} \gamma_{exp_{V_m}}(s_i\sigma') & if\ \forall e \in exp_{V_m}.[\![e]\!]_{\rho_{s_i}} = [\![e]\!]_{\rho_{s_{i+1}}} \\ s_i\gamma_{exp_{V_m}}(s_{i+1}\sigma') & if\ \exists e \in exp_{V_m}.[\![e]\!]_{\rho_{s_i}} \neq [\![e]\!]_{\rho_{s_{i+1}}} \end{cases}$$

where σ' is an infinite sequence of states, $[\![e]\!]_{\rho_{s_i}}$ for $e \in exp_{V_m}$ is the result of evaluating an boolean expression e according to an environment ρ_{s_i}, and $i \geq 0$.

Note that Definition 10 itself does not impose any restriction on the set exp_{V_m} except that boolean expressions in the exp_{V_m} should be expressions over the monitored variables V_m. For value abstraction to be useful, exp_{V_m} should be related to requirement properties $prop_{req}$.

Definition 11 (Valid Value Abstraction) *Value abstraction $\gamma_{exp_{V_m}}$ is valid with regard to the set of requirement properties $prop_{req}$ if and only if*

$$\forall j \geq 0. \Big(\forall e \in exp_{V_m}.[\![e]\!]_{\rho_{s_j}} = [\![e]\!]_{\rho_{s_{j+1}}} \longrightarrow \forall p \in prop_{req}.[\![p]\!]_{\rho_{s_j}} = [\![p]\!]_{\rho_{s_{j+1}}}\Big)$$

Definition 11 indicates that the removed states must *not* affect evaluation of requirement properties. The remaining states, however, may or may not affect evaluation result. In other words, valid value abstraction is *sound*, but not complete.

So far, we have not decided how to set exp_{V_m} with regard to $prop_{req}$. In one extreme end, exp_{V_m} can be a set of entire requirement properties, i.e., $exp_{V_m} = prop_{req}$. In this case, the abstract view taken by the monitor is equal to a sequence of fail/safe flags obtained by evaluating requirement properties. In the other extreme end, exp_{V_m} can be an empty set. In this case, no states are removed by value abstraction. We need to decide a point between these two extreme ends for setting exp_{V_m} with regard to $prop_{req}$ satisfying Definition 11.

For example, suppose that a requirement property is

$$p_{req} = ((3 < x) \wedge (x < 10)) \vee (y > 2) \wedge (z < 10)$$

One exp_{V_m} satisfying Definition 11 with regard to p_{req} is

$$exp_{V_m} = \{(3 < x) \wedge (x < 10),\ z < 10\}$$

With this exp_{V_m}, for example, states updating x from 4 to 9 or states updating z from 1 to 9 are removed by value abstraction because these states do not change evaluation result of $(3 < x) \wedge (x < 10)$ or $z < 10$.

In practice, value abstraction is closely related to reduce monitoring overhead. To apply value abstraction, a probe need to test $\forall e \in exp_{V_m}.[\![e]\!]_{\rho_{s_i}} = [\![e]\!]_{\rho_{s_{i+1}}}$ (see Definition 10) whenever a monitored variable is updated. Therefore, we have to consider both property evaluation overhead and probe's test overhead to reduce the overall monitoring overhead. Section 5.3 describes how Java-MaC implements value abstraction.

5 Overview of the Java-MaC

The monitoring and checking (MaC) framework has been designed to ensure that the execution of a real-time system is consistent with its requirements at run-time. Java-MaC instruments Java bytecodes and monitors/checks the correctness of the Java program's execution with regard to given formal requirement specification. The structure of the Java-MaC architecture is shown in Fig 2. The architecture includes two main phases: *static phase* and *run-time phase*. A formal specification consists of a low-level implementation dependent specification and a high-level requirement specification. A low-level specification language is called Primitive Event Definition Language (PEDL). A high-level specification language is called Meta Event Definition Language (MEDL). From a target program and PEDL/MEDL specifications, 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 a given formal requirement specification. More detail on Java-MaC can be found in[4,5]

Section 5.1 describes static phase of Java-MaC. Section 5.2 describes run-time phase of Java-MaC. Finally, Section 5.3 explains implementation of value abstraction in Java-MaC.

5.1 Static Phase

Java-MaC has three static phase components: an *instrumentor*, a *PEDL compiler*, and a *MEDL compiler*. A PEDL compiler compiles a PEDL script 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.

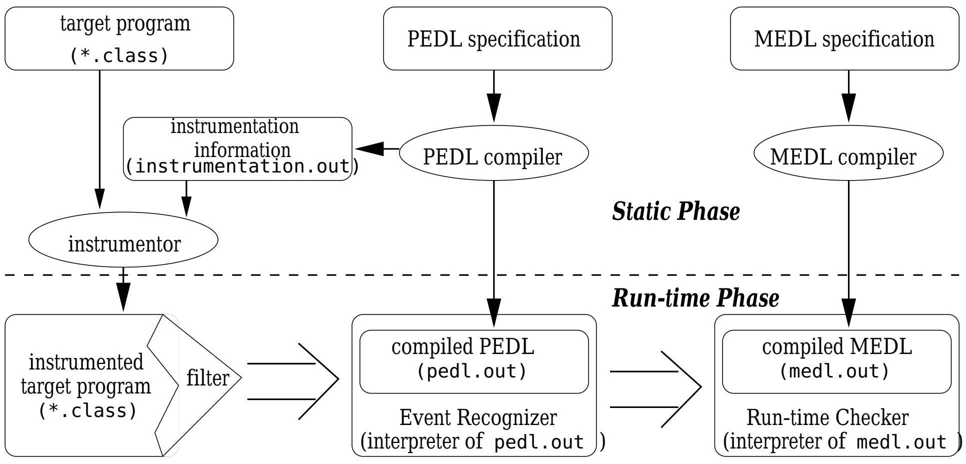


Fig. 2. Structure of Java-MaC

In addition, a PEDL compiler generates exp_{V_m} for value abstraction from a PEDL script and puts exp_{V_m} into `instrumentation.out`. A Java-MaC instrumentor takes a Java bytecode (`*.class`) and instrumentation information (`instrumentation.out`) containing a list of monitored variables/methods and exp_{V_m} generated from a PEDL script. Based on these inputs, the Java-MaC instrumentor inserts a filter consisting of probes into the target bytecode. Each probe contains a routine testing whether $\forall e \in exp_{V_m}.[\![e]\!]_{\rho_{s_i}} = [\![e]\!]_{\rho_{s_{i+1}}}$ as well as a routine for sending variable updates to an event recognizer.

A MEDL compiler compiles a MEDL script into an abstract syntax tree (`medl.out`) which is evaluated by a run-time checker at run-time. More details of instrumentor and PEDL/MEDL compilers are in [5].

5.2 Run-time Phase

During the run-time phase, the instrumented target program is executed while being monitored and checked with respect to a 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 variables and send snapshots of program execution states to the event recognizer. Furthermore, a filter tests $\forall e \in exp_{V_m}.[\![e]\!]_{\rho_{s_i}} = [\![e]\!]_{\rho_{s_{i+1}}}$ to decide whether a snapshot should be sent to the event recognizer or not. 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 requirement specification. The execution history is captured from a sequence of events sent by the event recognizer. The connection among these run-time components can be made through one of socket, FIFO file, or user defined communication methods.

5.3 Implementation of Value Abstraction

Java-MaC sets exp_{V_m} as a set of *simple expressions* obtained from event and condition definitions in a PEDL script.[5] A simple expression consists of one variable, a comparison operator, and a constant (ex. $x < 5.4$). For this set of exp_{V_m}, the amount of computation performed by probes for testing $\forall e \in exp_{V_m}.[\![e]\!]_{\rho_{s_i}} = [\![e]\!]_{\rho_{s_{i+1}}}$ is very little because comparison between a variable and a constant is computationally cheap; probes do not perform any boolean operations or numerical operations for the test.

Construction of exp_{V_m} from a PEDL script is as follows.

(i) Extract simple expressions se_{v_ij} for each monitored variable v_i from event definitions and condition definitions in a PEDL script.

(ii) Unless either
- there exists a *non*-simple expression containing monitored variable v_i
- there exists `update(`v_i`)` in event/condition definitions [6]

put se_{v_ij} into exp_{V_m}.

If exp_{V_m} contains se_{v_ij}, apply value abstraction whenever v_i is updated. Otherwise, send v_i to an event recognizer whenever v_i is updated. When value abstraction is applied, a probe monitoring a variable v_i tests all $se_{v_ij} \in exp_{V_m}$ whenever v_i is updated. If any simple expression se_{v_ij} fails the test, i.e. there exists an se_{v_ij} which has changed its value according to update of v_i, the probe sends a new value of v_i. Otherwise, not.

Consider the following example.

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

The variable `x` is used only in simple expressions. The filter keeps the truth values of each of the three simple expressions (`3 < x`, `x < 10`, and `x > 5`) that involve `x` and sends an update only when it changes one of the expression values as in Figure 3. Variables `y` and `z` are used in an expression which is not a simple expression. Thus, every update of `y` and `z` will be sent to the event recognizer. Whether value abstraction can be applied to a variable x or not can be decided by scanning a PEDL script by a PEDL compiler.

6 Example: the Sieve of Eratosthenes

We illustrate monitoring overhead and effectiveness of value abstraction using the Sieve of Eratosthenes program. The Sieve of Eratosthenes generates prime numbers. The algorithm of the Sieve for generating prime numbers less than

[5] We assume that all of event/condition definitions in a PEDL script are used to build requirement properties in a MEDL script.

[6] `update(`v_i`)` becomes `true` whenever v_i is updated.

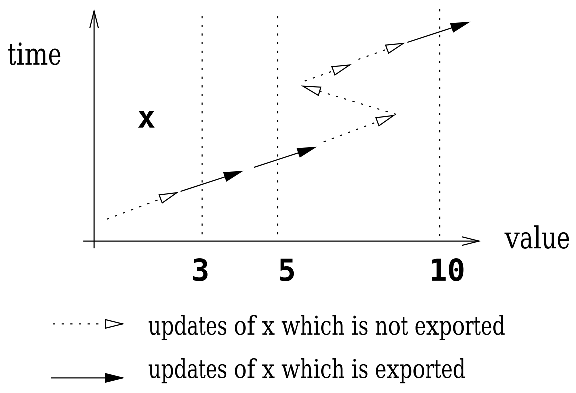

time
x
value
3
5
10
updates of x which is not exported
updates of x which is exported

Fig. 3. Example of updating `x`

or equal to n can be described as follows.

> Make a list of all the integers less than or equal to n (and greater than one). Strike out the multiples of all primes less than or equal to $\sqrt{n}$, then the numbers that are left are the primes.[7]

We would like to monitor and check whether there exists a prime number between 99990 and 100000. For that purpose, an event `foundPrime` is defined in lines 5 to 7 of Figure 4. The experiment shows that there exists one prime between 99990 and 100000.

```
01:MonScr
02:  export event foundPrime;
03:  monobj int SieveMain.sa.numTested;
04:  monobj int SieveMain.sa.numPrimes;
05:  event foundPrime = update(SieveMain.sa.numPrimes) when
06:                          (99990 <= SieveMain.sa.numTested
07:                          && SieveMain.sa.numTested <= 100000);
08:end
```

Fig. 4. PEDL script for checking the existence of prime between 99990 and 100000

Figure 5 shows a Java code for the Sieve of Eratosthenes. An integer being tested is declared as `numTested` in line 14. `numPrimes` declared in line 15 indicates the total number of prime numbers upto `numTested`. Main code in `execute()` from line 22 to line 42 contains a nested loop. Lines 30 to 41 form an outer loop which increases `numTested` one by one. Lines 33 to 35 make an inner loop which divides `numTested` with prime numbers less than or equal to $\sqrt{\texttt{numTested}}$. Lines 37 to 40 store `numTested` as a prime number and increase `numPrimes` by 1 if there is no prime number which can divide `numTested`.

The computational complexity of the Sieve program(Figure 5) is $O(n\sqrt{n})$ (the outer loop takes $O(n)$ and the inner loop takes $O(\sqrt{n})$) where n is a max-

[7] It is well-known mathematical fact that if there exists a prime greater than $\sqrt{n}$ which divides n, there exists a prime less than or equal to $\sqrt{n}$ which divides n, too.

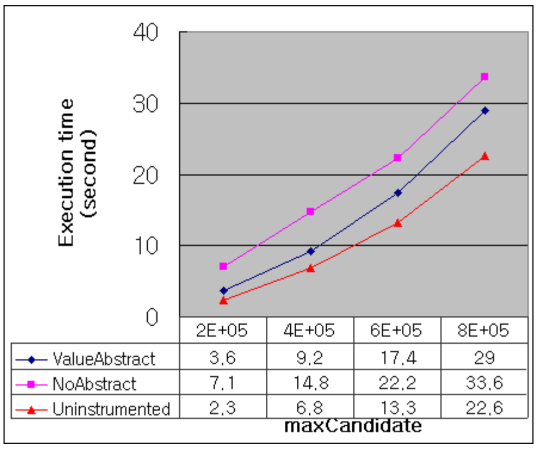
```
public class SieveMain{
    Sieve sa;
    SieveMain() { sa = new Sieve();}
    public static void main(String[] args) {
        SieveMain sm = new SieveMain();
        sm.sa.initialize( Integer.parseInt(args[0]));
        sm.sa.execute();
    }
}
class Sieve {
    public Sieve sa;
    public int primes[];
    public int maxCandidate;
    public int numTested;       // current number being tested
    public int numPrimes;       // number of primes found

    public void initialize(int i) {
        maxCandidate = i;
        primes = new int[maxCandidate];
    }

    public void execute() {
        int k = 0;
        int sqrt_i=0;
        boolean flag = false;

        primes[0] = 1;
        primes[1] = 2;
        numPrimes = 2;
        for (numTested = 3; numTested <= maxCandidate; numTested++) {
            k = 1;
            sqrt_i = (int)(Math.sqrt(i));
            for (flag = true; k < numPrimes && flag; k++)
                if ( primes[k] <= sqrt_i && numTested % primes[k] == 0)
                    flag = false;

            if (flag) {
                numPrimes++;
                primes[numPrimes - 1] = numTested;
            }
        }
    }
}
```

Fig. 5. The code of the Sieve of Eratosthenes

imum number to check. The Sieve program sends around 1.08×n snapshots to an event recognizer (n×`numTested` + 0.08×`numPrimes`).

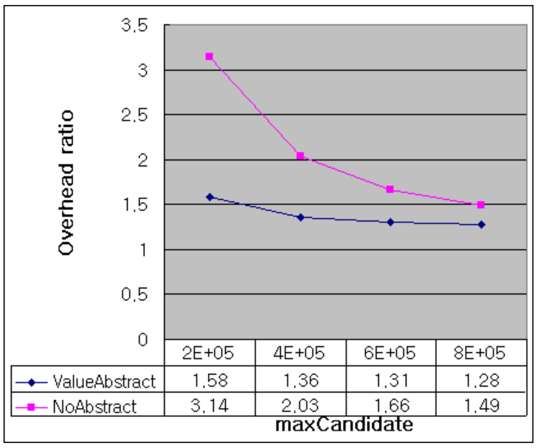
We performed the experiment using Linux 2.2 machine (2X 550 Mhz PIII, 1GB memory, IBM JDK 1.3.1) and Windows 2000 machine (1.4Ghz PIV, 512MB memory, Sun JDK 1.3.1). The sieve program runs on Linux machine. Event recognizer runs on Windows 2000 machine. The bottom line of Figure 6.a) shows the execution time of the uninstrumented Sieve program. Testing integers from 1 to 200000 takes 2.3 seconds. Testing integers from 1 to 800000, however, takes 22.6 seconds. The frequency of taking snapshots decreases as n increases because of $O(n\sqrt{n})$ computational complexity of the Sieve program, which decreases the overhead ratio by Java-MaC as n increases. Figure 6.b) shows the overhead ratios of NoAbstract, and ValueAbstract. NoAbstract slows down the Sieve program 3.1 times when n is 200000. The overhead ratio decreases to 1.5 times when n increases to 800000. ValueAbstract reduces the overhead 73% compared to the overhead of NoAbstract when n is 200000. As n increases, the bottleneck of event evaluation diminishes, which decreases the amount of reduction by ValueAbstract.

a)



	2E+05	4E+05	6E+05	8E+05
ValueAbstract	3.6	9.2	17.4	29
NoAbstract	7.1	14.8	22.2	33.6
Uninstrumented	2.3	6.8	13.3	22.6

b)



	2E+05	4E+05	6E+05	8E+05
ValueAbstract	1.58	1.36	1.31	1.28
NoAbstract	3.14	2.03	1.66	1.49

Fig. 6. Overhead to the Sieve of Eratosthenes (a) Execution time (b) Overhead ratio

7 Conclusion

We have described theoretical study as well as practical implication of computational characteristics of run-time monitoring. We define the class of properties run-time monitoring can evaluate accurately; a class of monitorable properties is a strict subset of a class of safety properties. In addition, we proved that evaluating properties written in process algebra takes intractable time by transforming 3SAT problem to the trace validity problem. Finally, a sound abstraction of the target program's execution is developed which filters out states unrelavant to the requirement properties. Java-MaC uses value abstraction to reduce the property evaluation overhead. Value abstraction is a quite effective technique to reduce overhead, as we have seen in the case study of the Sieve of Erathosthenes, where monitoring overhead is reduced 73% by value abstraction.

This paper focuses on mathematical treatment of monitoring overhead. Equally important topic is to measure and limit the range of monitoring overhead. This topic is important because safety real-time systems require the upper bound of worst case execution time for their guarantee of correct execution. We are investigating real-time JVM as we upgrade Java-MaC to incorporate steering features. Another topic is to take more advantage of value abstraction. Current implementation of Java-MaC tests only simple expressions for value abstraction in order to minimize probe's overhead. More aggressive way of applying value abstraction, i.e. setting exp_{V_m} not only simple expressions, but more compound ones seems promising. Also, we may analyze execution history statistically to maximize states reduction of value abstraction.

References

[1] P. Brémond-Grégoire, I. Lee, and R. Gerber. ACSR: An Algebra of Communicating Shared Resources with Dense Time and Priorities. In *Proc. of CONCUR '93*, 1993.

[2] Sarah E. Chodrow and Mohamed G. Gouda. Implementation of the Sentry System. In *Software – Practice and Experience*. John Wiley & Sons, Ltd., April 1995.

[3] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2002.

[4] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-Mac: a run-time assurance tool for java programs. *Runtime Verification Paris France*, 2001.

[5] Moonjoo Kim. *Information Extraction for Run-time Formal Analysis*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2001.

[6] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

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

[8] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2000.