Runtime Safety Analysis of Multithreaded Programs

Koushik Sen [*]
Dept. of Computer Science
University of Illinois at Urbana
ksen@uiuc.edu

Grigore Roşu [†]
Dept. of Computer Science
University of Illinois at Urbana
grosu@uiuc.edu

Gul Agha [‡]
Dept. of Computer Science
University of Illinois at Urbana
agha@uiuc.edu

ABSTRACT

Foundational and scalable techniques for runtime safety analysis of multithreaded programs are explored in this paper. A technique based on vector clocks to extract the causal dependency order on state updates from a running multithreaded program is presented, together with algorithms to analyze a multithreaded computation against safety properties expressed using temporal logics. A prototype tool implementing our techniques, is also presented, together with examples where it can predict safety errors in multithreaded programs from successful executions of those programs. This tool is called Java MultiPathExplorer (JMPaX), and available for download on the web. To the best of our knowledge, JMPaX is the first tool of its kind.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification

Keywords

LTL, predictive analysis, safety analysis, runtime monitoring, vector clock, multithreaded program, JMPaX, Java

[*]Supported in part by the Defense Advanced Research Projects Agency (Contract numbers: F30602-00-2-0586 and F33615-01-C-1907) and the ONR Grant N00014-02-1-0715.

[†]Supported in part by joint NSF/NASA grant CCR-0234524

[‡]Supported in part by the Defense Advanced Research Projects Agency (Contract numbers: F30602-00-2-0586 and F33615-01-C-1907) and the ONR Grant N00014-02-1-0715.

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.

1. INTRODUCTION

The purpose of this paper is to investigate foundational, scalable techniques for runtime safety analysis of multithreaded programs, i.e., programs in which several execution threads communicate with each other via shared variables and synchronization points, as well as to introduce a prototype tool, called Java MultiPathExplorer (JMPaX – see Figure 1), based on our foundational techniques, which can reveal errors in multithreaded programs that are hard or impossible to detect otherwise. The user of JMPaX specifies safety properties of interest, using a past time temporal logic, regarding the global state of a multithreaded program, which is already assumed in compiled form, calls an instrumentation script which automatically instruments the executable multithreaded program to emit relevant state update events to an external observer, and finally runs the program on any JVM and analyzes the safety violation messages reported by the observer. A particularly appealing aspect of our approach is that, despite the fact that a single execution, or interleaving, of a multithreaded program can be observed, a comprehensive analysis of *all possible executions* is performed; a possible execution is any execution which does not violate the observed causal dependency partial order on state update events. Thus, tools built on our techniques, including JMPaX, have the ability to *predict* safety violation errors in multithreaded programs by observing successful executions.

The work in this paper falls under the area recently called *runtime verification* [11, 10], a major goal of which is to combine testing and formal methods techniques. Testing scales well, and is by far the most used technique in practice to validate software systems. By merging testing and temporal logic specification, we aim to achieve the benefits of both approaches, while avoiding some of the pitfalls of ad hoc testing and the complexity of full-blown theorem proving and model checking. Of course, there is a price to be paid in order to obtain a scalable technique: the loss of coverage. The suggested framework can only be used to examine single execution traces, and therefore can not be used to *prove* a system correct. However, a single execution trace typically contains much more information than what appears at first sight. In this paper, we show how one can analyze all the other multithreaded executions that are hidden behind a particular observed execution. Our work is based on the belief that software engineers are willing to trade coverage for scalability, so our goals are to provide tools that use formal methods techniques in a lightweight manner, use unmodified programming languages or under-

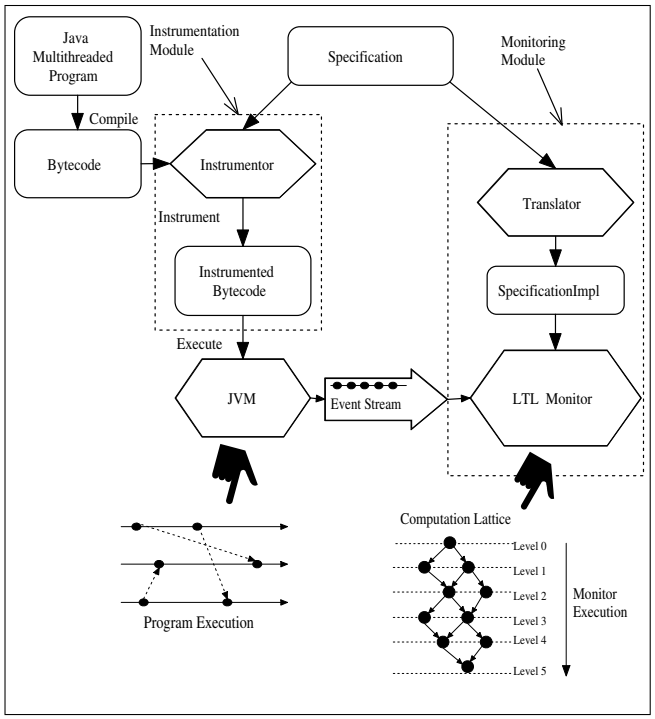

Figure 1: JMPaX Architecture

lying executional engines (such as JVMs), are completely automatic, implement very efficient algorithms and eventually find *many* errors in programs. A longer term goal is to explore the use of conformance with a formal specification to achieve error recovery. The idea is that a predicted failure may trigger an error-avoidance or recovery action in the monitored program.

The closest works in spirit to ours are NASA's PathExplorer (PaX) and its Java instance JPaX [10, 9], which is a runtime verification system developed at NASA Ames, and UPENN's MaC and its instance Java MaC [14, 15]. It is actually the latter's limitations that motivated us to pursue our current research. The major limitation of these systems with regards to safety analysis is that they only analyze the observed run. Therefore, they can only detect existing errors in current executions; they do not have the ability to predict possible errors from successful runs. To be more precise in our claim, let us consider a real-life example where JMPaX was able to detect a violation of a safety property from a single execution of the program. However, the likelihood of detecting this bug only by monitoring the observed run, as JPaX and Java-MAC do, is very low. The example consists of a two threaded program to control the landing of an airplane. It has three variables `landing`, `approved`, and `radio`; their values are 1 when the *plane is landing*, *landing has been approved*, and *radio signal is live*, and 0 otherwise. The safety property that we want to verify is "If the plane has started landing, then it is the case that landing has been approved and since the approval the radio signal has never been down." As shown in Subsection 3.1, this property can be formally written in our extension of past time linear temporal logic as the formula

$$\uparrow\texttt{landing} \rightarrow [\texttt{approved}, \downarrow\texttt{radio})_s.$$

The code snippet for a naive implementation of this control program is given as follows:

```
int landing = 0, approved = 0, radio = 1;
void thread1(){
   askLandingApproval();
   if(approved==1){
       print("Landing approved");
       landing = 1;
       print("Landing started");
   } else {
       print("Landing not approved");
   }
}

void askLandingApproval(){
   if(radio==0) approved = 0;
   else approved = 1;
}

void thread2(){
   while(radio){checkRadio();} }

void checkRadio(){
   randomly change value of radio;
}
```

The above code uses some dummy functions, namely `askLandingApproval` and `checkRadio`, which can be implemented in their entirety in a real scenario. The program has a serious problem which cannot be detected easily from a single run. The problem is as follows. Suppose the plane has received approval for landing and just before it started landing the radio signal went off. In this situation, the plane must abort landing. But this situation will very rarely arise in an execution: namely, when `radio` is set to 0 between the approval of landing and the start of actual landing. So a simple observer will not probably detect the bug. However, note that even if the radio goes off after the landing has started, JMPaX can still construct a possible run in which radio goes off between landing and approval. Thus JMPaX will be able to predict the safety violation from a single successful execution of the program. This example shows the power of our runtime verification technique as compared to JPaX and Java-MaC.

Other related approaches include model checking [6], especially Java bytecode model checking [8], and debugging of distributed systems. It is important to observe that, unlike model checking where all possible code interleavings are analyzed, in our approach to runtime safety analysis one actually runs the program and extracts causal dependencies among updates of the multithreaded program state, and then analyzes all possible interleavings that do not violate the causal dependency. At the expense of a lower coverage, our approach analyzes a significantly lower amount of thread interleavings than a typical model checker would normally do, so it scales up better. The safety properties that we analyze are more general than the simpler state predicates that are typically considered in the literature on debugging distributed systems (see for example [19, 4, 3]). We allow any past time linear temporal logic formula built on state predicates, so our safety properties can refer to the entire past history of states. An important practical aspect of our algorithm is that, despite the fact that there can be a potentially exponential number of runs (in the length of the runs), they can all be analyzed *in parallel*, by generating and traversing the computation lattice extracted from the

observed multithreaded execution on a level-by-level basis. The relevant information regarding the previous levels can be encoded compactly, so those levels do not need to be stored, thus allowing the memory to be reused.

We can think of at least three major contributions of the work presented in this paper. First, we nontrivially extend the runtime safety analysis capabilities of systems like JPaX and Java Mac, by providing the ability to predict safety errors from successful executions; we are not aware of any other efforts in this direction. Second, we underlie the foundations of relevant causality in multithreaded systems with shared variables and synchronization points, which one can use to instrument multithreaded programs to emit to external observers a causal dependency partial relation on global state updates via relevant events timestamped with appropriate vector clocks; this is done in Section 2, where, due to its foundational aspect, all the proofs of the claimed results are provided. Finally, a modular implementation of a prototype runtime analysis system, JMPaX, is given, showing that, despite their theoretical flavor, all the concepts presented in the paper are in fact quite practical and can lead to new scalable verification tools.

2. RELEVANT CAUSALITY IN MULTITHREADED SYSTEMS

We consider multithreaded systems in which several threads communicate with each other via a set of shared variables. The theme of this paper is to show how such a system can be analyzed for safety by an external observer that obtains relevant information about the system from messages sent by the system after appropriate instrumentation. The safety formulae refer to sets of shared variables, so these messages contain update information about those variables. A crucial observation here is that some variable updates can causally depend on others. For example, if a thread writes a variable x and then another thread writes y due to a statement $y = x + 1$, then the update of y *causally depends* upon the update of x. In this section we present an algorithm which, given an executing multithreaded system, generates appropriate messages to be sent to an external observer. The observer, in order to perform its more elaborated safety analysis, extracts the state update information from such messages together with the causality partial order order among the updates.

Formally, given n threads p_1, p_2, ..., p_n, a *multithreaded execution* is abstracted as a sequence of events $e_1 e_2 \dots e_r$, each belonging to one of the n threads and being of type either *internal* or *read* or *write* of a shared variable. We use e_i^j to represent the j-th event generated by thread p_i since the start of its execution. From now on in this section we assume an arbitrary but fixed multithreaded execution. When the process or the position of an event is not important then we can refer to the event generically, such as e, e', etc.; we may write $e \in p_i$ when event e is generated by thread p_i. Let S be the set of shared variables. There is an immediate notion of *variable access precedence* for each shared variable $x \in S$: we say that e *x-precedes* e', written $e <_x e'$, if and only if e and e' are variable access events (reads or writes) to the same variable x, and e "happens before" e'; this "happen-before" relation can be easily realized by keeping a counter for each shared variable which is increased by each access to it. Let $\mathcal{E}$ be the set of all the events of a multithreaded execution, and let $\prec$ be the partial order on $\mathcal{E}$ defined as follows:

- $e_i^k \prec e_i^l$ if $k < l$;
- $e \prec e'$ if there is some $x \in S$ such that $e <_x e'$ and atleast one of e, e' is a write.
- $e \prec e''$ if $e \prec e'$ and $e' \prec e''$.

We write $e||e'$ when it is not the case that $e \prec e'$ or $e' \prec e$. A partial order on events $\prec$ defined above is called a *multithreaded computation* associated with the original multithreaded execution. As shown in Subsection 3.3, synchronization can be treated very elegantly by generating appropriate read/write events, so that the notion of multithreaded computation as defined above is as general as currently needed. Note that the original multithreaded execution was used only to provide a total ordering on the read/write accesses of each shared variable.[1] A permutation of all the events e_1, e_2, ..., e_r which does not violate the multithreaded computation is called a *consistent multithreaded run*, or simply, a *multithreaded run*.

Intuitively, $e \prec e'$, read as e' *causally depends upon* e, if and only if e occurred before e' in the given multithreaded execution and a change of their order does not generate a consistent multithreaded run. We argue that the notion of multithreaded computation defined above is the *weakest assumption* that an omniscient observer of the multithreaded execution can make about the program. Intuitively, this is because an external observer *cannot disregard* the order in which the same variable is modified and used within the observed execution, because this order can be part of the intrinsic logic of the multithreaded program. However, multiple consecutive reads of the same variable can be permuted, and the particular order observed in the given execution is not critical; it can be, for example, a result of a particular thread scheduling algorithm. By allowing an observer to analyze *multithreaded computations* rather than just *multithreaded run*, one gets the benefit of not only properly dealing with potential reordering of delivered messages (for example, due to using multiple channels in order to reduce the monitoring overhead), but also of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling.

Not all the variable in S are needed to evaluate the safety formula to be checked. To minimize number of messages sent to an observer, and for technical reasons discussed later, we consider a subset $\mathcal{R} \subseteq \mathcal{E}$ of *relevant events*. Then we define the *$\mathcal{R}$-relevant causality* on $\mathcal{E}$ as the relation $\lhd := \prec \cap (\mathcal{R} \times \mathcal{R})$, so that $e \lhd e'$ if and only if $e, e' \in \mathcal{R}$ and $e \prec e'$. We provide a technique based on *vector clocks* [7, 17] that correctly implements the relevant causality relation.

Let V_i be an n-dimensional vector of natural numbers for thread p_i, for each $1 \leq i \leq n$, and let V_x^a and V_x^w be two additional n-dimensional vectors for each shared variable x; we call the former *access vector clock* and the latter *write vector clock*. All the vector clocks are initialized to 0 at the beginning of computation. For two n-dimensional vectors we say that $V \leq V'$ if and only if $V[j] \leq V'[j]$ for all $1 \leq j \leq n$, and we say that $V < V'$ iff $V \leq V'$ and there is some $1 \leq j \leq n$ such that $V[j] < V'[j]$; also, $\max\{V, V'\}$ is

[1] One could have defined a multithreaded computation more abstractly but less intuitively, by starting with a total order $<_x$ on the subset of events accessing each shared variable x.

the vector with $\max\{V, V'\}[j] = \max\{V[j], V'[j]\}$ for each $1 \leq j \leq n$. Whenever a thread p_i with current vector clock V_i processes event e_i^k, the following vector clock algorithm is executed:

1. if e_i^k is relevant, i.e., if $e_i^k \in \mathcal{R}$, then
   $V_i[i] \leftarrow V_i[i] + 1$
2. if e_i^k is a read of a variable x then
   $V_i \leftarrow \max\{V_i, V_x^w\}$
   $V_x^a \leftarrow \max\{V_x^a, V_i\}$
3. if e_i^k is a write of a variable x then
   $V_x^w \leftarrow V_x^a \leftarrow V_i \leftarrow \max\{V_x^a, V_i\}$
4. if e_i^k is relevant then
   send message $\langle e_i^k, i, V_i\rangle$ to observer.

Then the following crucial results hold:

LEMMA 1. *After event e_i^k is processed by thread p_i,*

a. *$V_i[j]$ equals the number of relevant events of p_j that causally precede e_i^k; if $j = i$ and e_i^k is relevant then this number also includes e_i^k;*
b. *$V_x^a[j]$ equals the number of relevant events of p_j that causally precede the most recent event that accessed (read or wrote) x; if $i = j$ and e_i^k is a relevant read or write of x event then this number also includes e_i^k;*
c. *$V_x^w[j]$ equals the number of relevant events of p_j that causally precede the most recent write event of x; if $i = j$ and e_i^k is a relevant write of x then this number also includes e_i^k.*

THEOREM 2. *If $\langle e, i, V\rangle$ and $\langle e', j, V'\rangle$ are two messages sent by our algorithm, then $e \lhd e'$ if and only if $V[i] \leq V'[i]$. If i and j are not given, then $e \lhd e'$ if and only if $V < V'$.*

In a summary, the above theorem states that the vector clock algorithm correctly implements causality in multithreaded programs. The detailed proofs of the above results are given in [18].

Consider what happens at the observer's site. The observer receives messages of the form $\langle e, i, V\rangle$ in any possible order. We let $\mathcal{R}$ denote the set of received relevant events, which we simply call *events* in what follows. By using Theorem 2, the observer can infer the causal dependency between the relevant events emitted by the multithreaded system. Inspired by similar definitions at the multithreaded system's [2], we define the important notions of relevant multithreaded computation and run as follows. A *relevant multithreaded computation*, simply called *multithreaded computation* from now on, is the partial order on events that the observer can infer, which is nothing but the relation $\lhd$. A *relevant multithreaded run*, also simply called *multithreaded run* from now on, is any permutation of the received events which *does not violate* the multithreaded computation. Our purpose in this paper is to check safety requirements against *all* (relevant) multithreaded runs of a multithreaded system.

We assume that the relevant events are only writes of shared variables that appear in the safety formulae to be monitored, and that these events contain a pair of the name of the corresponding variable and the value which was written to it. We call these variables *relevant variables*. Note that events can change the state of the multithreaded system as seen by the observer; this is formalized next. A *relevant program state*, or simply a *program state* is a map from relevant variables to concrete values. Any permutation of events generates a sequence of program states in the obvious way, however, not all permutations of events are valid multithreaded runs. A program state is called *consistent* if and only if there is a multithreaded run containing that state in its sequence of generated program states. We next formalize these concepts.

For a given permutation of (relevant) events in $\mathcal{R}$, say $e_1 e_2 \ldots e_{|\mathcal{R}|}$, we can let e_i^k denote the k-th event of thread p_i for each $1 \leq i \leq n$. Then the relevant program state after the events $e_1^{k_1}, e_2^{k_2}, ..., e_n^{k_n}$ is called a *relevant global multithreaded state*, or simply a *relevant global state* or even just *state*, and is denoted by $\Sigma^{k_1 k_2 \ldots k_n}$. A state $\Sigma^{k_1 k_2 \ldots k_n}$ is called *consistent* if and only if for any $1 \leq i \leq n$ and any $l_i \leq k_i$, it is the case that $l_j \leq k_j$ for any $1 \leq j \leq n$ and any l_j such that $e_j^{l_j} \lhd e_i^{l_i}$. Let Σ^{K_0} be the *initial* global state, $\Sigma^{00\ldots0}$. An important observation is that $e_1 e_2 \ldots e_{|\mathcal{R}|}$ is a multithreaded run if and only if it generates a sequence of global states $\Sigma^{K_0}\Sigma^{K_1}\ldots\Sigma^{K_{|\mathcal{R}|}}$ such that each Σ^{K_r} is consistent and for any two consecutive Σ^{K_r} and $\Sigma^{K_{r+1}}$, K_r and K_{r+1} differ in exactly one index, say i, where the i-th element in K_{r+1} is larger by 1 than the i-th element in K_r. For that reason, we will identify the sequences of states $\Sigma^{K_0}\Sigma^{K_1}\ldots\Sigma^{K_{|\mathcal{R}|}}$ as above with multithreaded runs, and simply call them *runs*. We say that Σ *leads-to* Σ', written $\Sigma \rightsquigarrow \Sigma'$, when there is some run in which Σ and Σ' are consecutive states. The set of all consistent global states together with the relation $\rightsquigarrow$ forms a *lattice*. The lattice has n mutually orthogonal axis representing each thread. For a state $\Sigma^{k_1 k_2 \ldots k_n}$, we call $k_1 + k_1 + \cdots k_n$ its *level*. A *path* in the lattice is a sequence of consistent global states on increasing level, where the level increases by 1 between any two consecutive states in the path. Therefore, a run is just a path starting with $\Sigma^{00\ldots0}$ and ending with $\Sigma^{r_1 r_2 \ldots r_n}$, where r_i is the total number of events of thread i for each $1 \leq i \leq n$. Therefore, a multithreaded computation can be seen as a lattice; we call this lattice a *computation lattice*.

EXAMPLE 1. Suppose that one wants to monitor some safety property of the multithreaded program below. The program involves relevant variables x, y and z:

```
Initially: x = −1; y = 0; z = 0;
thread T1{                thread T2{
  ...                       ...
  x++;                      z = x + 1;
  ...                       ...
  y = x + 1;                x++;
  ...                       ...
}                         }
```

The ellipses (...) indicate code that is not relevant, i.e., that does not access the variables x, y and z. This multithreaded program, after appropriate instrumentation, sends messages to an observer whenever the relevant variables are updated. A possible execution of the program to be sent to the observer, described in terms of relevant variable updates, can be

$$\{x = -1, y = 0, z = 0\}, \{x = 0\}, \{z = 1\}, \{y = 1\}, \{x = 1\}$$

The first message to observer sends the initial state of the whole system as a set of variable-value pairs. The second event is generated when the value of x is incremented by the first thread. The above execution corresponds to the

sequence of program states

$$(-1,0,0),\ (0,0,0),\ (0,0,1),\ (0,1,1),\ (1,1,1)$$

where the tuple $(-1,0,0)$ denotes the state in which $x = -1, y = 0, z = 0$. Following the vector clock algorithm, we can deduce that the observer will receive the multithreaded computation in Figure 2 which generates the computation lattice shown in the same figure.

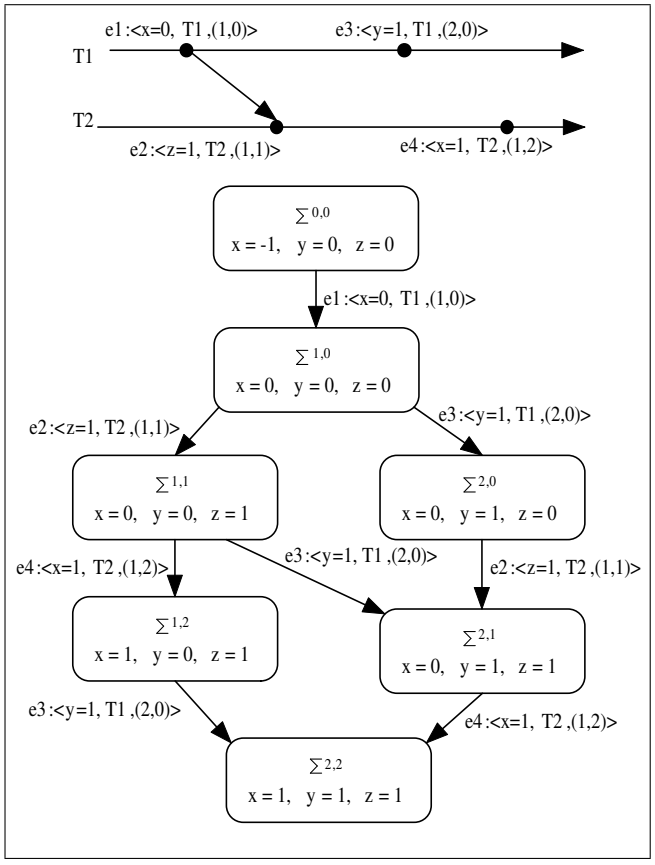

Figure 2: Computation lattice and three runs.

Notice that the observed multithreaded execution corresponds to just one particular multithreaded run out of the three possible. We will show that it is often possible that the observed run does not violate any safety property, but the run nevertheless shows that there are other possible runs that are not safe. We will propose an algorithm that will detect safety violations in any possible run, even though the violation was not revealed by the particular observed run. An appealing aspect of our algorithm is that, despite the fact that there can be a potentially exponential number of runs (in the maximum width of a level), they can all be analyzed *in parallel*, by generating and traversing the lattice on a level-by-level basis; the previous levels are not needed, so memory can be reused.

3. MULTITHREADED SAFETY ANALYSIS

In this section, we first introduce the past time temporal logic that we use to express safety properties, then we recall an algorithm to monitor such properties efficiently against a single run, and finally we show how this algorithm non-trivially extends to monitoring multithreaded computations given as partial orders.

3.1 Safety in Temporal Logics

We use past time Linear Temporal Logic (*ptLTL*)[16] to express our safety properties. Our choice of past time linear temporal logic is motivated by two considerations:

1. It is powerful enough to express safety properties of concurrent systems;
2. The monitors for a safety formula written in *ptLTL* are very efficient; they perform linearly in the size of the formula in the worst case [12].

Now we briefly introduce the basic notions of *ptLTL* , and describe some new operators that are particularly useful for runtime monitoring. The syntax of *ptLTL* is given as follows:

$$\begin{array}{lll} F ::= & true \mid false \mid a \in A \mid \neg F \mid F\ op\ F & \text{Propositional ops} \\ & \odot F \mid \Diamond F \mid \Box F \mid F \mathcal{S}_s F \mid F \mathcal{S}_w F & \text{Standard operators} \\ & \uparrow F \mid \downarrow F \mid [F,F)_s \mid [F,F)_w & \text{Monitoring ops} \end{array}$$

where *op* are the standard binary operators, namely $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and $\odot F$ should be read as "previously", $\Diamond F$ as "eventually in the past", $\Box F$ as "always in the past", $F_1 \mathcal{S}_s F_2$ as "F_1 strong since F_2", $F_1 \mathcal{S}_w F_2$ as "F_1 weak since F_2", $\uparrow F$ as "start F", $\downarrow F$ as "end F", $[F_1, F_2)_s$ as "strong interval F_1, F_2", and $[F_1, F_2)_w$ as "weak interval F_1, F_2".

The logic is interpreted on a finite sequence of states or a run. If $\rho = s_1 s_2 \ldots s_n$ is a run then we let ρ_i denote the prefix run $s_1 s_2 \ldots s_i$ for each $1 \leq i \leq n$. The semantics of the different operators is given in Table 1.

The monitoring operators $\uparrow$, $\downarrow$, $[,)_s$, and $[,)_w$ were introduced in [12, 15]. These operators have been found to be very intuitive and useful in specifying properties for runtime monitoring. Informally, $\uparrow F$ is true if and only if F *starts* to be true in the current state, $\downarrow F$ is true if and only if F *ends* being true in the current state, and $[F_1, F_2)_s$ is true if and only if F_2 was never true since the last time F_1 was observed to be true, including the state when F_1 was true; the interval operator has a strong and a weak version. For example, if *boot* and *down* are predicates on the state of a web server to be monitored, say for the last 24 hours, then $[boot, down)_s$ is a property stating that the server *was* rebooted recently and the since then it was not down, while $[boot, down)_w$ say that server was not unexpectedly down recently, meaning that it was either not down at all recently or it was rebooted recently and since then it was not down.

In runtime monitoring, we start the process of monitoring from the point the first event is generated and it continues as long as the events are generated. Thus given a *ptLTL* formula F we check whether $\rho_i \models F$ for all $1 \leq i \leq n$, where $\rho = s_1 s_2 \ldots s_n$ is a finite run constructed from the events.

3.2 Checking Safety Against a Single Run

We describe an algorithm for monitoring the multithreaded execution or *the observed run* of a multithreaded computation, which is just one path in the computation lattice, and illustrate it through an example. This algorithm is a modified version of the algorithm presented in [12]. The algorithm computes the boolean value of the formula to be monitored, by recursively evaluating the boolean value of

$\rho \models true$	is true for all ρ,
$\rho \models a$	iff a holds in the state s_n,
$\rho \models \neg F$	iff $\rho \not\models F$,
$\rho \models F_1 \ op \ F_2$	iff $\rho \models F_1$ and/or/implies/iff $\rho \models F_2$, when op is $\wedge / \vee / \rightarrow / \leftrightarrow$,
$\rho \models \odot F$	iff $\rho' \models F$, where $\rho' = \rho_{n-1}$ if $n > 1$ and $\rho' = \rho$ if $n = 1$,
$\rho \models \Diamond F$	iff $\rho_i \models F$ for some $1 \leq i \leq n$,
$\rho \models \Box F$	iff $\rho_i \models F$ for all $1 \leq i \leq n$,
$\rho \models F_1 \mathcal{S}_s F_2$	iff $\rho_j \models F_2$ for some $1 \leq j \leq n$ and $\rho_i \models F_1$ for all $1 \leq i \leq n$,
$\rho \models F_1 \mathcal{S}_w F_2$	iff $\rho \models F_1 \mathcal{S}_s F_2$ or $\rho \models \Box F_1$,
$\rho \models \uparrow F$	iff $\rho \models F$ and it is not the case that $\rho \models \odot F$,
$\rho \models \downarrow F$	iff $\rho \models \odot F$ and it is not the case that $\rho \models F$,
$\rho \models [F_1, F_2)_s$	iff $\rho_j \models F_1$ for some $1 \leq j \leq n$ and $\rho_i \not\models F_2$ for all $j \leq i \leq n$,
$\rho \models [F_1, F_2)_w$	iff $[F_1, F_2)_s$ or $\rho \models \Box \neg F_2$,

Table 1: Semantics of *ptLTL*

each of its subformulae in the current state. In the process it also uses the boolean value of certain subformulae evaluated in the previous state. Next we define this recursive function *eval*. The recursive nature of the temporal operators in *ptLTL* enables us to define the boolean value of a formula in the current state in terms of its boolean value in the previous state and the boolean value of its subformulae in the current state. For example we can define:

$\rho \models \Diamond F$	iff $\rho \models F$ or ($n > 1$ and $\rho_{n-1} \models \Diamond F$)
$\rho \models \Box F$	iff $\rho \models F$ and ($n > 1$ implies $\rho_{n-1} \models \Box F$)
$\rho \models F_1 \mathcal{S}_s F_2$	iff $\rho \models F_2$ or ($n > 1$ and $\rho \models F_1$ and $\rho_{n-1} \models F_1 \mathcal{S}_s F_2$)
$\rho \models F_1 \mathcal{S}_w F_2$	iff $\rho \models F_2$ or ($\rho \models F_1$ and ($n > 1$ and $\rho_{n-1} \models F_1 \mathcal{S}_w F_2$))
$\rho \models [F_1, F_2)_s$	iff $\rho \not\models F_2$ and ($\rho \models F_1$ or ($n > 1$ and $\rho_{n-1} \models [F_1, F_2)_s$))
$\rho \models [F_1, F_2)_w$	iff $\rho \not\models F_2$ and ($\rho \models F_1$ or ($n > 1$ implies $\rho_{n-1} \models [F_1, F_2)_w$))

These definitions correspond to the code for the cases of the operators $\Diamond$, $\Box$, $\mathcal{S}_s$, $\mathcal{S}_w$, $[,)_s$, and $[,)_w$ in the function *eval*. The function *op(f)* returns the operator of the formula f, *binary(op(f))* returns *true* if *op(f)* is binary, *unary(op(f))* returns *true* if *op(f)* is true, *left(f)* returns the left subformula of f, *right(f)* returns the right subformula of f, and *subformula(f)* returns the subformula of f.

```
boolean
eval(Formula f, State s, array now, array pre, int index){
  if binary(op(f)) then{
    lval ← eval(left(f), now, pre, index);
    rval ← eval(right(f), now, pre, index); }
  else if unary(op(f)) then
    val ← eval(subformula(f), now, pre, index);
  case(op(f)) of{
    p : return p(s); true : return true; false : return false;
    op : return rval op lval; ¬ : return not val;
    S_s, S_w : now[++index] ← lval or rval;
      return (pre[index] and lval) or rval;
    [,)_s, [,)_w :
      now[++index] ← (not rval) and (pre[index] or lval);
      return (not rval) and (pre[index] or lval);
    ↑ : now[++index] ← val;
      return (not pre[index]) and val;
    ↓ : now[++index] ← val;
      return pre[index] and (not val);
    □ : now[++index] ← val; return pre[index] and val;
    ◇ : now[++index] ← val; return pre[index] or val;
    ⊙ : now[++index] ← val; return pre[index];
  }
}
```

Here, the *pre* array passed as an argument contains the boolean values of all subformulae in the previous state, that will be required in the current state. While the *now* array, after the evaluation of *eval* function, will contain the boolean values of all subformulae in the current state that will be required in the next state. Note, here the *now* array is passed as reference, and its value is set in the function *eval*. The function *eval*, however, cannot be used to evaluate the boolean value of a formula for the first state in a run, as the recursion handles the case $n = 1$ in a different way. We define the function *init* to handle this special case as follows:

```
boolean init(Formula f, State s, array now, int index){
  if binary(op(f)) then{
    lval ← init(left(f), now, index);
    rval ← eval(right(f), now, index); }
  else if unary(op(f)) then
    val ← init(subformula(f), now, index);
  case(op(f)) of{
    p : return p(s); true : return true; false : return false;
    op : return rval op lval; ¬ : return not val;
    S_s : now[++index] ← rval; return rval;
    S_w : now[++index] ← lval or rval; return lval or rval;
    [,)_s : now[++index] ← (not rval) and lval;
      return (not rval) and lval;
    [,)_w : now[++index] ← (not rval); return (not rval);
    ↑, ↓ : now[++index] ← val; return false;
    □, ◇, ⊙ : now[++index] ← val; return val;
  }
}
```

For a given formula f, we define the function *monitor*, that at each iteration, consumes an event in the run, generates the next state from that event, and evaluates the formula for the state generated:

```
monitor(Formula f, Run r = e_1 e_2 ... e_n){
  State state ← {}; array now, pre;
  state ← update(state, e_1);
  val ← init(f, state, now, 0);
```

```
  if (not val) then output('property violated');
  for i = 2 to n do{
    pre ← now;
    state ← update(state, e_i);
    val ← eval(f, state, now, pre, 0);
    if (not val) then output('property violated');
  }
}
```

In the initialization phase, the *state* variable is created from the event e_1. The *now* array is then calculated by calling the function *init* on the current *state*. After the calculation the result of *init* is checked for falsity, and an error message is issued if the result is false. Otherwise, the main loop is started. The main loop goes through the run, starting from the second event. At each iteration, *now* is copied to *pre*, the current state is generated by consuming an event from the run, the formula f is evaluated in the current state using the function *eval*, the result of evaluation is tested for falsity and an error message is generated if the result is false.

The time complexity of this algorithm is $\Theta(mn)$, where m is the size of the original formula and n is the length of the run. However, memory required by the algorithm[2] is $2m'$, m' being the number of temporal and monitor operators in the formula.

We now go back to the EXAMPLE 1. Suppose that one want to monitor the safety property $(x > 0) \rightarrow [(x = 0), y > z)_s$ on that program. The formula states that "if $(x > 0)$, then $(x = 0)$ has been true in the past, and since then $(y > z)$ was always false."

For the possible execution or the observed run of the program mentioned in Section 2, we have the following sequence of global states,

$$(-1, 0, 0),\ (0, 0, 0),\ (0, 0, 1),\ (0, 1, 1),\ (1, 1, 1)$$

where the tuple $(-1, 0, 0)$ denotes the state in which $x = -1, y = 0, z = 0$. The formula is satisfied in all the states of the sequence and so we say that the property specified by the formula is not violated by the given run. However, another possible run of the same computation is,

$$\{x = -1, y = 0, z = 0\}, \{x = 0\}, \{y = 1\}, \{z = 1\}, \{x = 1\}$$

This run corresponds to the sequence of states

$$(-1, 0, 0),\ (0, 0, 0),\ (0, 1, 0),\ (0, 1, 1),\ (1, 1, 1)$$

The formula is clearly violated in the last state of this sequence. This is because, $x > 0$ in the 5th state. This means that from 2nd state, in which $x = 0$, up to 5th state $y > z$ must be false. However, $y > z$ in the 3rd state. This violates the formula. Therefore, the monitoring algorithm that considers only the observed run presented in this subsection fails to detect this violation. In the next subsection we propose an algorithm that will detect such a potential bug from the original successful run.

3.3 Checking Safety Against All Runs

The algorithm, presented in the previous subsection, can only monitor a single run. As noticed earlier, monitoring one run may not reveal a bug that might be present in other possible runs. Our algorithm removes this limitation by monitoring all the possible runs of a multithreaded computation. The major hurdle in monitoring all possible runs is that the number of possible runs can be exponential in the length of the computation. We avoid this problem in our algorithm by traversing the computation lattice level by level. The events are generated by the algorithm presented in Section 2. The monitoring module consumes these events one by one, and monitors the safety formula on the computation lattice constructed from the events. We now describe the monitoring module in more details.

The monitoring module maintains a queue of events. Whenever an event arrives from the running multithreaded program, it *enqueues* it in the event queue (Q). The module also maintains a set of global states (*CurrentLevel*), that are present in the current level of the lattice. For each event e in the event queue, it tries to construct a global state from the set of states in the current level and the event e. If the global state is created successfully it is added to the set of global states (*NextLevel*) for the next level of the lattice. Once a global state in the current level becomes *unnecessary*, it is removed from the set of global states in the current level. When the set of global states in the current level becomes empty, we say that the set of global states for the next level is *complete*. At that time the module checks the safety formula (by calling *monitorAll(NextLevel)*) for the set of states in the next level. If the formula is not violated it marks the set of global states for the next level as the set of states for the current level, removes unnecessary events from the event queue, and restarts the iteration. The pseudocode for the process is given below:

```
for each (e ∈ Q){
  if ∃s ∈ CurrentLevel s.t. isNextState(s, e) then
    NextLevel ← addToSet(NextLevel, createState(s, e));
  if isUnnecessary(s) then remove(s, CurrentLevel);
  if isEmpty(CurrentLevel) then{
    monitorAll(NextLevel);
    CurrentLevel ← NextLevel; NextLevel ← {};
    Q ← removeUnnecessaryEvents(CurrentLevel, Q);
  }
}
```

Every global state s contains the value of all relevant shared variables in the program, a n-dimensional vector clock *VC(s)* to represent the latest events from each thread that resulted in that global state, and a vector of boolean values called *flags*. Each component of *flags* is initially set to *false*. Here the predicate *isNextState(s,e)*, checks if the event e can convert the state s to a state s' in the next level of the lattice. The pseudocode for the predicate is given below:

```
boolean isNextState(s, e){
  i ← threadId(e);
  if VC(s)[i] = VC(e)[i] + 1 then{
    flags(s)[i] = true;
    if (∀ 1 ≤ j ≤ n, j ≠ i) VC(s)[j] ≥ VC(e)[j] then
      return true; else return false;}
  else return false;
}
```

where n is the number of threads, *threadId(e)* returns the

[2]Here we assume that the recursive version is properly converted into an iterative algorithm using cps transform.

index of the thread that generated the event e, *VC(s)* returns the vector clock of global state s, *VC(e)* returns the vector clock of event e, and *flags(s)* returns the vector *flags* associated with s. Note, here *flags(s)[i]* is set to *true* if $VC(s)[i] = VC(e)[i] + 1$. This means that e is the only event from thread i that *can possibly* take state s to a state s' in the next level. When all the components of the vector *flags(s)* become *true*, we say that the state s is *unnecessary*. Thus the function *isUnnecessary(s)* checks if $(\forall\, 1 \leq i \leq n)$ *flags(s)*$[i] =$ *true*, where n is the number of threads.

The function *createState(s,e)* creates a new global state s', where s' is a possible global state that can result from s after the event e. For the purpose of monitoring we maintain, with every global state, a set of *pre* arrays called *PreSet*, and a set of *now* arrays called *NowSet*. In the function *createState* we set the *PreSet* of s' with the *NowSet* of s. The pseudocode for *createState* is as follows:

```
State createState(s, e){
  s' ← new copy of s;
  j ← threadId(e); VC(s')[j] ← VC(s)[j] + 1;
  for i = 1 to n {flags(s')[i] ← false;}
  state(s')[var(e) ← value(e)]; return s';
  PreSet(s') ← NowSet(s);
}
```

Here *state(s')* returns the value of all relevant shared variables in state s', *var(e)* returns the name of the relevant variable that is written at the time of event e, *value(e)* is the value that is written to *var(e)*, and *state(s')*$[var(e) \leftarrow value(e)]$ means that in *state(s')*, *var(e)* is updated with *value(e)*.

The function *addToSet(NextLevel,s)* adds the global state s to the set *NextLevel*. If s is already present in *NextLevel*, it updates the existing states' *PreSet* with the union of the existing states' *PreSet* and the *PreSet* of s. Two global states are same if their vector clocks are equal. The function *removeUnnecessaryEvents(CurrentLevel,Q)* removes from Q the events that cannot contribute to the construction of any state at the next level. To do so, it creates a vector clock V_{min} whose each component is the minimum of the corresponding component of the vector clocks of all the global states in the set *CurrentLevel*. It then removes all the events in Q whose vector clocks are less than or equal to V_{min}. This function makes sure that we do not store the unnecessary events.

The function *monitorAll* performs the actual monitoring of the formula. In this function, for each state s in the set *NextLevel*, we invoke the function *eval* (as discussed in the previous section) on s, for each *pre* array in the set *PreSet*. If *eval* returns *false*, we issue a `'property violation'` warning. The *now* array that resulted from the invocation of *eval* is added to the set *NowSet* of s. The pseudocode for the function *monitorAll* is given as follows:

```
monitorAll(NextLevel){
  for each s ∈ NextLevel{
    for each pre ∈ PreSet(s){
      now ← {}; result ← eval(f, s, now, pre, 0);
      if not result then output('property violated');
      NowSet(s) ← NowSet(s) ∪ {now};}
  }
}
```

If the multithreaded program has synchronization blocks, then we introduce, during instrumentation, a dummy shared variable that is read whenever we enter the synchronization block and is written when we exit the block. This makes sure that all the events in one execution of the block are causally dependent on the events in another execution of the same block, so that the interleaving between them becomes impossible.

Here the size of each *pre* array or *now* array is m', where m' is the number of temporal operators in the formula. Therefore, the size of the set *PreSet* or the set *NowSet* can be atmost $2^{m'}$. This implies that the memory required for each state in the lattice is $O(2^{m'})$. If the maximum width of the lattice is w, then the total memory required by the program is $O(w2^{m'})$. The time taken by the algorithm at each iteration is $O(w2^m)$, where m is the size of the formula. However, if the threads in a program have very few dependency or synchronization points, then the number of valid permutations of the events can be very large, and therefore the width of the lattice can become large. To handle those situations we can add a parameter to the algorithm which specifies the maximum width of the lattice. Then, if the number of states in a level of the lattice becomes larger than the maximum width, the algorithm can be modified to consider only the most *probable states* in the level. We can specify different heuristics to calculate the most probable states in a given level of the lattice.

4. IMPLEMENTATION

We have implemented our monitoring algorithm, in a tool called Java Multi PathExplorer (JMPaX)[1], which has been designed to monitor multithreaded Java programs. The current implementation, see Figure 1, is written in Java and it assumes that all the shared variables of the multithreaded program are static variables of type `int`. The tool has two main modules, the *instrumentation* module and the *monitoring* module. The instrumentation program, named `instrument`, takes a specification file, a port number, and a list of class files as command line arguments. An example of such command is

```
java instrument spec server 7777 A.class B.class
C.class
```

where the specification file `spec` contains a list of named formulae. The specification for the example discussed in Section 2 looks as follows:

```
F = (A.x > 0) -> [(A.x = 0),(A.y > A.z))s
```

where `A` is the class containing the shared variables `x`, `y` and `z` as static fields. The program `instrument` instruments the classes, provided in the argument, as follows:

i) It adds *access* and *write* vector clocks as static fields for each shared variable;

ii) It adds code to create a vector clock whenever a thread is created;

iii) For each read and write access of the shared variables in the class files, it adds codes to update the vector clocks according to the algorithm mentioned in Section 2;

iv) It adds codes to send messages to the `server` at the port number 7777 for all writes of relevant variables.

To do so, the `instrument` program extracts the relevant variables from the specification file.

The instrumentation module uses BCEL [5] Java library to modify Java class files. We use the BCEL library to get a better handle for a Java classfile. It enables us to insert vector clocks as static member fields in a class, that is otherwise not possible with the tool JTrek [13]. We also make the update of vector clocks associated with a read or write, atomic through `synchronization`. For this we need to add Java bytecode both before and after the instructions `getstatic` and `putstatic`, that access the shared variables. This task is easier in BCEL as compared to JTrek.

A *translator*, which is part of monitoring module, reads the specification and generates a single Java class file, named `SpecificationImpl.class`. The monitoring module starts a server to listen events from the instrumented program, parses them, enqueues them in a queue, executes `translator` to generate `SpecificationImpl.class`, dynamically loads the class `SpecificationImpl.class`, and starts monitoring the formulae on the queue of events. It issues a warning whenever a formula is violated.

One of the test cases that we have implemented is the landing example described in Section 1. JMPaX was able to detect violation of a safety property from a single execution of the program. The safety property that we verified was:

$$\uparrow\texttt{landing} \rightarrow [\texttt{approved}, \downarrow\texttt{radio})_s.$$

From a single execution of the code in which the radio went off after the landing, JMPaX constructed a possible run in which radio goes off between landing and approval, and hence it detected the safety violation. This example shows the power of our runtime verification technique.

5. CONCLUSION AND FUTURE WORK

We have investigated the problem of runtime analysis of multithreaded systems from a fundamental perspective. We have developed scalable techniques for extracting relevant events and their causal dependency from an executing multithreaded program. We have proposed and implemented algorithms to check safety properties against the computation lattice of a multithreaded computation. We have also briefly presented our prototype tool Java MultiPathExplorer, abbreviated JMPaX, which, at our knowledge, is the first tool that can predict violations of safety properties expressed in temporal logics from correct executions of multithreaded programs. We have also shown that, despite the fact that our safety properties can refer to any state in the past and that there is a potentially exponential number of multithreaded runs to be analyzed, one does not need to actually store the previous states; one can analyze all the multithreaded runs in parallel, by traversing the computation lattice top down, level-by-level.

Three major contributions have been made. First, we have nontrivially extended the capabilities of systems like JPaX and Java Mac, by providing the ability to *predict safety errors* from successful executions; we regard safety prediction as an important trade-off towards avoiding the inherent complexity of full-blown theorem proving and model checking; we are not aware of any other efforts in this direction. Second, we have defined the notion of relevant causality in multithreaded systems with shared variables and synchronization points and we have provided a technique of implementing relevant causality based on vector clocks. Finally, we have implemented a modular prototype runtime analysis system, JMPaX; modularity comes from the fact that its instrumentation module can be used together with other computation lattice analysis tools, while its safety computation analysis module can be used in any event based setting, for example a distributed system. In fact, we intend to soon extend our work to analyzing arbitrary distributed systems at runtime for not only safety but also other properties of interest. There are also plans on developing a predictive analysis runtime environment for both multithreaded and distributed systems, as well as developing a GUI for JMPaX that would make it easy to use and understand by ordinary software engineers. Since our work is partly sponsored by NASA, we also intend to soon use JMPaX on real-world NASA-related large applications.

6. REFERENCES

[1] Java Multi PathEXplorer, March 2003. `http://fsl.cs.uiuc.edu/jmpax/`.

[2] H. W. Cain and M. H. Lipasti. Verifying sequential consistency using vector clocks. In *Proceedings of the 14th annual ACM symposium on Parallel algorithms and Architectures*, pages 153–154. ACM, 2002.

[3] C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[4] R. Cooper and K. Marzullo. Consistent detection of global predicates. *ACM SIGPLAN Notices*, 26(12):167–174, 1991. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.

[5] M. Dahm. Byte code engineering with the bcel api. Technical Report B-17-98, Freie Universit at Berlin, Institut für Informatik, April 2001.

[6] J. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[7] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed debugging*, pages 183–194. ACM, 1988.

[8] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Apr. 2000.

[9] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[10] K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. IEEE, 2001.

[11] K. Havelund and G. Roşu. *Runtime Verification 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001. Proceedings of a *Computer Aided Verification (CAV'01)* satellite workshop.

[12] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Proceedings Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, pages 342–356, 2002.

[13] JTrek Compaq Corp. `www.digital.com/java/download/jtrek/`.

[14] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[15] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[16] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer-Verlag N.Y., Inc., 1995.

[17] F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier science, 1989.

[18] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. Technical Report UIUCDCS-R-2003-2334, University of Illinois at Urnaba Champaign, April 2003.

[19] S. D. Stoller. Detecting global predicates in distributed systems with clocks. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, pages 185–199, 1997.

*