Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Online Efficient Predictive Safety Analysis of Multithreaded Programs

Koushik Sen and Grigore Roşu and Gul Agha
Department of Computer Science,
University of Illinois at Urbana-Champaign.
{ksen,grosu,agha}@cs.uiuc.edu

Received: date / Revised version: date

Abstract. We present an automated and configurable technique for runtime safety analysis of multithreaded programs which is able to *predict* safety violations from successful executions. Based on a formal specification of safety properties that is provided by a user, our technique enables us to automatically instrument a given program and create an observer so that the program emits *relevant* state update events to the observer and the observer checks these updates against the safety specification. The events are stamped with *dynamic vector clocks*, enabling the observer to infer a *causal partial order* on the state updates. All event traces that are consistent with this partial order, including the actual execution trace, are then analyzed *online* and *in parallel*. A warning is issued whenever one of these potential trace violates the specification. Our technique is scalable and can provide better coverage than conventional testing but its coverage need not be exhaustive. In fact, one can trade-off scalability and comprehensiveness: a *window* in the state space may be specified allowing the observer to infer some of the *more likely* runs; if the size of the window is 1 then only the actual execution trace is analyzed, as is the case in conventional testing; if the size of the window is ∞ then all the execution traces consistent with the actual execution trace are analyzed.

1 Introduction

In multithreaded systems, threads can execute concurrently communicating with each other through a set of shared variables, yielding an inherent potential for subtle errors due to unexpected interleavings. Both rigorous and light-weight techniques to detect errors in multithreaded systems have been extensively investigated. Rigorous techniques include formal methods, such as model checking and theorem proving, which by exploring –directly or indirectly– all possible thread interleavings, guarantee that a formal model of the system satisfies its safety requirements. Unfortunately, despite impressive recent advances, the size of systems for which model checking or automatic theorem proving is feasible remains rather limited. As a result, most system builders continue to use light-weight techniques such as testing to identify bugs in their implementations.

There are two problems with software testing. First, testing is generally done in an *ad hoc* manner: the software developer must hand-translate the requirements into specific dynamic checks on the program state. Second, test coverage is often rather limited, covering only some execution paths: if an error is not exposed by a particular test case then that error is not detected. To mitigate the first problem, software often includes dynamic checks on a system's state in order to identify problems at run-time. To ameliorate the second problem, many techniques increase test coverage by developing test-case generation methods that generate test cases which may reveal potential errors with high probability [6,15,26].

Based on experience with related techniques and tools, namely Java PathExplorer (JPaX) [12] and its sub-system Eagle [2], we have proposed in [22,23] an alternative approach, called *predictive runtime analysis*. The essential idea of this analysis technique is as follows. Suppose that a multithreaded program has a safety error, such as a violation of a temporal property, a deadlock, or a data-race. As in testing, we execute the program on some carefully chosen input (a test case). Suppose that the error is not revealed during a particular execution, i.e., the execution is *successful* with respect to that bug. If one regards the execution of a program as a flat, sequential trace of events or states, as in NASA's JPaX system [12], University of Pennsylvania's Java-MaC [14], Bell Labs' PET [11], Nokia's Third Eye framework [16] inspired by Logic Assurance system [24], or the commercial analysis systems Temporal Rover and DBRover [7–9], then there is not much left to do to find the error except to run another, hopefully better, test case. However, by observing the execution trace in a smarter way, namely as a causal dependency partial order on state updates, we can predict errors that may potentially occur in other possible runs of the multithreaded program.

Our technique merges testing and formal methods to obtain some of the advantages of both while avoiding the pitfalls of *ad hoc* testing and the complexity of full-blown formal verification. Specifically, we develop a *runtime verification* technique for safety analysis of multithreaded systems that can be tuned to analyze a number of traces that are consistent with an actual execution of the program. Two extreme instances of our technique involve checking all or one of the variant traces:

– If all traces are checked then it becomes equivalent to online model checking of an abstract model of the program, called the *multithreaded computation lattice*, extracted from the actual execution trace of the program, like in POTA [19] or JMPaX [22].
– If only one trace is considered, then our technique becomes equivalent to checking just the actual execution of the multithreaded program, as is done in testing or like in other runtime analysis tools like MaC [14] and PaX [12,2].

In general, depending on the application, one can configure a window within the state space to be explored which, intuitively speaking, provides a causal *distance* from the observed execution within which all traces are exhaustively verified. We call such a window a *causality cone*. An appealing aspect of our technique is that all these traces can be analyzed *online*, as the events are received from the running program, and *in parallel*. The worst case cost of such an analysis is proportional to both the size of the window and the size of the state space of the monitor.

There are three important interrelated components in our runtime verification technique. Our algorithm synthesizes these components automatically from the safety specification:

Instrumentor. The code instrumentor, based on the safety specification, entirely automatically adds code to emit events when *relevant* state updates occur.

Observer. The observer receives the events from the instrumented program as they are generated, enqueues them and then builds a configurable abstract model of the system, known as a computation lattice, on a layer-by-layer basis.

Monitor. As layers are completed, the monitor checks them against the safety specification and then discards them.

The concepts and notions presented in this paper have been experimented and tested on JMPaX 2.0, a prototype monitoring system for Java programs that we have built. JMPaX 2.0 extends its predecessor JMPaX in at least four non-trivial novel ways:

– The technical notion of *dynamic vector clock* is introduced, which allows us to properly deal with the dynamic creation and destruction of threads.
– The variables that are shared between threads need not be static: an automatic instrumentation technique has been devised that detects automatically when a variable is shared.
– The notion of *cone heuristic*, or *global state window*, is introduced. The cone heuristic enables us to increase the runtime efficiency by analyzing the most likely states in the computation lattice and tune how comprehensive we wish to be.
– The runtime prediction paradigm used is independent of the safety formalism, in the sense that it allows the user to specify any safety property whose bad prefixes can be expressed as a non-deterministic finite automaton (NFA).

Part of this work was presented at the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04) [23].

2 Monitors for Safety Properties

Safety properties are a very important, if not the most important, class of properties that one should consider in monitoring. This is because once a system violates a safety property, there is no way to continue its execution to satisfy the safety property later. Therefore, a monitor for a safety property can precisely say at runtime when the property has been violated, so that an external recovery action can be taken. From a monitoring perspective, what is needed from a safety formula is a succinct representation of its *bad prefixes*, which are finite sequences of states leading to a violation of the property. Therefore, one can abstract away safety properties by languages over finite words.

Automata are a standard means to succinctly represent languages over finite words. In what follows we define a suitable version of automata, called *monitor*, with the property that it has a "bad" state from which it never gets out:

Definition 1. Let $\mathcal{S}$ be a finite or infinite set, that can be thought of as the set of states of the program to be monitored. Then an $\mathcal{S}$-*monitor* or simply a *monitor*, is a tuple $\mathcal{M}on = \langle \mathcal{M}, m_0, b, \rho \rangle$, where

– $\mathcal{M}$ is the set of states of the monitor;
– $m_0 \in \mathcal{M}$ is the initial state of the monitor;
– $b \in \mathcal{M}$ is the *final state* of the monitor, also called *bad state*; and

– $\rho\colon \mathcal{M} \times \mathcal{S} \to 2^{\mathcal{M}}$ is a non-deterministic transition relation with the property that $\rho(b, \Sigma) = \{b\}$ for any $\Sigma \in \mathcal{S}$.

Sequences in $\mathcal{S}^\star$, where ϵ is the empty one, are called *(execution) traces*. A trace π is said to be a *bad prefix in $\mathcal{M}on$* iff $b \in \rho(\{m_0\}, \pi)$, where $\rho\colon 2^{\mathcal{M}} \times \mathcal{S}^\star \to 2^{\mathcal{M}}$ is recursively defined as $\rho(M, \epsilon) = M$ and $\rho(M, \pi\Sigma) = \rho(\rho(M, \pi), \Sigma)$, where $\rho\colon 2^{\mathcal{M}} \times \mathcal{S} \to 2^{\mathcal{M}}$ is defined as $\rho(\{m\} \cup M, \Sigma) = \rho(m, \Sigma) \cup \rho(M, \Sigma)$ and $\rho(\emptyset, \Sigma) = \emptyset$, for all finite $M \subseteq \mathcal{M}$ and $\Sigma \in \mathcal{S}$.

$\mathcal{M}$ is not required to be finite in the above definition, but $2^{\mathcal{M}}$ represents the set of *finite* subsets of $\mathcal{M}$. In practical situations it is often the case that the monitor is *not* explicitly provided in a mathematical form as above. For example, a monitor can be a specific type of program whose execution is triggered by receiving events from the monitored program; its state can be given by the values of its local variables, and the bad state is a fixed unique state which once reached cannot be changed by any further events.

There are fortunate situations in which monitors can be *automatically generated* from formal specifications, thus requiring the user to focus on system's formal safety requirements rather than on low level implementation details. In fact, this was the case in all the experiments that we have performed so far. We have so far experimented with requirements expressed either in extended regular expressions (ERE) or various variants of temporal logics, with both future and past time. For example, [20,21] show coinductive techniques to generate minimal static monitors from EREs and from future time linear temporal logics, respectively, and [13,2] show how to generate dynamic monitors, i.e., monitors that generate their states on-the-fly, as they receive the events, for the safety segment of temporal logic. Note, however, that there may be situations in which the generation of a monitor may not be feasible, even for simple requirements languages. For example, it is well-known that the equivalent automaton of an ERE may be non-elementary larger in the worst case [25]; therefore, there exist relatively small EREs whose monitors cannot even be stored.

Example 1. Consider a reactive controller that maintains the water level of a reservoir within safe bounds. It consists of a water level reader and a valve controller. The water level reader reads the current level of the water, calculates the quantity of water in the reservoir and stores it in a shared variable `w`. The valve controller controls the opening of a valve by looking at the current quantity of water in the reservoir. A very simple and naive implementation of this system contains two threads: `T1`, the valve controller, and `T2`, the water level reader. The code snippet is given in Fig. 1.

Here `w` is in some proper units such as mega gallons and `v` is in percentage. The implementation is poorly synchronized and it relies on ideal thread scheduling.

```
Thread T1:

while(true) {
  if(w > 18) delta = 10;
  else delta = -10;
  for(i=0; i<2; i++) {
    v = v + delta;
    setValveOpening(v);
    sleep(100);
  }
}
```

```
Thread T2:

while(true) {
  l = readLevel();
  w = calcVolume(l);
  sleep(100);
}
```

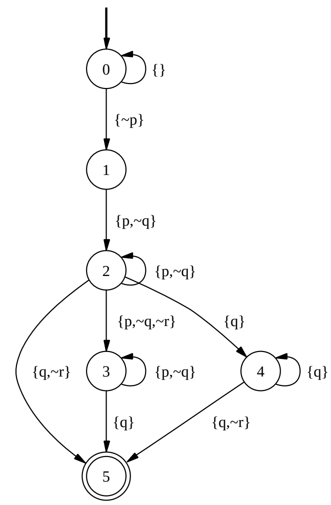

Fig. 1. Two threads (`T1` controls the valve and `T2` reads the water level) and a monitor.

A sample run of the system can be $\{w = 20, v = 40\}, \{w = 24\}, \{v = 50\}, \{w = 27\}, \{v = 60\}, \{w = 31\}, \{v = 70\}$. As we will see later in the paper, by a run we here mean a sequence of relevant variable writes. Suppose we are interested in a safety property that says "If the water quantity is more than 30 mega gallons, then it is the case that sometime in the past water quantity exceeded 26 mega gallons and since then the valve is open by more than 55% and the water quantity never went down below 26 mega gallon". We can express this safety property in two different formalisms: linear temporal logic (LTL) with both past-time and future-time operators, or extended regular expressions (EREs) for bad prefixes. The atomic propositions that we will consider are $p : (w > 26), q : (w > 30), r : (v > 55)$. The properties can be written as follows:

$$
\begin{aligned}
F_1 &= \Box(q \to ((r \wedge p)\mathcal{S} \uparrow p)) \\
F_2 &= \{\}^*\{\neg p\}\{p, \neg q\}^+ \\
&\quad (\{p, \neg q, \neg r\}\{p, \neg q\}^*\{q\} + \{q\}^*\{q, \neg r\})\{\}^*
\end{aligned}
$$

The formula F_1 in LTL ($\uparrow p$ is a shorthand for "p and previously not p") states that "It is always the case that if $(w > 30)$ then at some time in the past $(w > 26)$ started to be true and since then $(r > 55)$ and $(w > 26)$."

The formula F_2 characterizes the prefixes that make F_1 false. In F_2 we use $\{p, \neg q\}$ to denote a state where p and $\neg q$ holds and r may or may not hold. Similarly, $\{\}$ represents any state of the system. The monitor automaton for F_2 is given also in Fig. 1.

3 Multithreaded Programs

We consider multithreaded systems in which threads communicate with each other via shared variables. A crucial point is that some variable updates can causally depend on others. We will describe an efficient *dynamic vector clock* algorithm which, given an executing multithreaded program, generates appropriate messages to be sent to an external observer. Section 4 will show how the observer, in order to perform its more elaborated analysis, extracts the state update information from such messages together with the causality partial order.

3.1 Multithreaded Executions and Shared Variables

A multithreaded program consists of n threads t_1, t_2, ..., t_n that execute concurrently and communicate with each other through a set of shared variables. A *multithreaded execution* is a sequence of events $e_1 e_2 \ldots e_r$ generated by the running multithreaded program, each belonging to one of the n threads and having type *internal*, *read* or *write* of a shared variable. We use e_i^j to represent the j^{th} event generated by thread t_i since the start of its execution. When the thread or position of an event is not important we can refer to it generically, such as e, e', etc.; we may write $e \in t_i$ when event e is generated by thread t_i. Let us fix an arbitrary but fixed multithreaded execution, say $\mathcal{C}$, and let S be the set of all variables that were shared by more than one thread in the execution. There is an immediate notion of *variable access precedence* for each shared variable $x \in S$: we say e *x-precedes* e', written $e <_x e'$, iff e and e' are variable access events (reads or writes) to the same variable x, and e "happens before" e', that is, e occurs before e' in $\mathcal{C}$. This can be realized in practice by keeping a counter for each shared variable, which is incremented at each variable access.

3.2 Causality and Multithreaded Computations

Let $\mathcal{E}$ be the set of events occurring in $\mathcal{C}$ and let $\prec$ be the partial order on $\mathcal{E}$:

- $e_i^k \prec e_i^l$ if $k < l$;
- $e \prec e'$ if there is $x \in S$ with $e <_x e'$ and at least one of e, e' is a write;
- $e \prec e''$ if $e \prec e'$ and $e' \prec e''$.

We write $e||e'$ if $e \not\prec e'$ and $e' \not\prec e$. The tuple $(\mathcal{E}, \prec)$ is called the *multithreaded computation* associated with the original multithreaded execution $\mathcal{C}$. Synchronization of threads can be easily and elegantly taken into consideration by just generating dummy read/write events when synchronization objects are acquired/released, so the simple notion of multithreaded computation as defined above is as general as practically needed. A permutation of all events e_1, e_2, ..., e_r that does not violate the multithreaded computation, in the sense that the order of events in the permutation is consistent with $\prec$, is called a *consistent multithreaded run*, or simply, a *multithreaded run*.

A multithreaded computation can be thought of as the *most general assumption* that an observer of the multithreaded execution can make about the system without knowing what it is supposed to do. Indeed, an external observer simply *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 semantics 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. As seen in Section 4, by allowing an observer to analyze *multithreaded computations* rather than just *multithreaded executions*, one gets the benefit of not only properly dealing with potential reorderings of delivered messages (e.g., due to using multiple channels in order to reduce the monitoring overhead), but especially of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling.

3.3 Relevant Causality

Some of the variables in S may be of no importance at all for an external observer. For example, consider an observer whose purpose is to check the property "if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false"; formally, using the interval temporal logic notation in [13], this can be compactly written as $(x > 0) \to [y = 0, y > z)$. All the other variables in S except x, y and z are essentially irrelevant for this observer. To minimize the number of messages, like in [17] which suggests a similar technique but for distributed systems in which reads and writes are not distinguished, we consider a subset $\mathcal{R} \subseteq \mathcal{E}$ of *relevant events* and define the $\mathcal{R}$*-relevant causality* on $\mathcal{E}$ as the relation $\lhd := \prec \cap (\mathcal{R} \times \mathcal{R})$, so that $e \lhd e'$ iff $e, e' \in \mathcal{R}$ and $e \prec e'$. It is important to notice though that the other variables can also indirectly influence the relation $\lhd$, because they can influence the relation $\prec$. We next provide a technique based on *vector clocks* that correctly implements the relevant causality relation.

3.4 Dynamic Vector Clock Algorithm

We provide a technique based on *vector clocks* [10,3,18,1] that correctly and efficiently implements the relevant causality relation. Let $V : \mathit{ThreadId} \to \mathit{Nat}$ be a *partial* map from thread identifiers to natural numbers. We call such a map a *dynamic vector clock (DVC)* because its partiality reflects the intuition that threads are dynamically created and destroyed. To simplify the exposition and the implementation, we assume that each DVC V is a total map, where $V[t] = 0$ whenever V is not defined on thread t.

We associate a DVC with every thread t_i and denote it by V_i. Moreover, we associate two DVCs V_x^a and V_x^w with every shared variable x; we call the former *access DVC* and the latter *write DVC*. All the DVCs V_i are kept empty at the beginning of the computation, so they do not consume any space. For DVCs V and V', we say that $V \leq V'$ if and only if $V[j] \leq V'[j]$ for all j, and we say that $V < V'$ iff $V \leq V'$ and there is some j such that $V[j] < V'[j]$; also, $\max\{V, V'\}$ is the DVC with $\max\{V, V'\}[j] = \max\{V[j], V'[j]\}$ for each j. Whenever a thread t_i with current DVC V_i processes event e_i^k, the following algorithm $\mathcal{A}$ 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.

In the following discussion we assume a fixed number of threads n. In a program where threads can be created and destroyed dynamically, we only consider the threads that have causally affected the final values of the relevant variables at the end of the computation. For the above algorithm the following result holds:

Lemma 1. *After event e_i^k is processed by thread t_i*

(a) $V_i[j]$ equals the number of relevant events of t_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 t_j that causally precede the most recent event in $\mathcal{C}$ 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 t_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.

To prove the above lemma we introduce some useful formal notation and then state and prove the following two lemmas. For an event e_i^k of thread t_i, let $(e_i^k]$ be the indexed set $\{(e_i^k]_j\}_{1\leq j\leq n}$, where $(e_i^k]_j$ is the set $\{e_j^l \mid e_j^l \in t_j,\ e_j^l \in \mathcal{R},\ e_j^l \prec e_i^k\}$ when $j \neq i$ and the set $\{e_i^l \mid l \leq k,\ e_i^l \in \mathcal{R}\}$ when $j = i$. Intuitively, $(e_i^k]$ contains all the events in the multithreaded computation that causally precede or are equal to e_i^k.

Lemma 2. *With the notation above, for $1 \leq i, j \leq n$:*

1. $(e_j^{l'}]_j \subseteq (e_j^l]_j$ if $l' \leq l$;
2. $(e_j^{l'}]_j \cup (e_j^l]_j = (e_j^{\max\{l',l\}}]_j$ for any l and l';
3. $(e_j^l]_j \subseteq (e_i^k]_j$ for any $e_j^l \in (e_i^k]_j$; and
4. $(e_i^k]_j = (e_j^l]_j$ for some appropriate l.

Proof: *1.* is immediate, because for any $l' \leq l$, any event e_j^k at thread t_j preceding or equal to $e_j^{l'}$, that is one with $k \leq l'$, also precedes e_j^l.

2. follows by *1.*, because it is either the case that $l' \leq l$, in which case $(e_j^{l'}]_j \subseteq (e_j^l]_j$, or $l \leq l'$, in which case $(e_j^l]_j \subseteq (e_j^{l'}]_j$. In either case *2.* holds trivially.

3. There are two cases to analyze. If $i = j$ then $e_j^l \in (e_i^k]_j$ if and only if $l \leq k$, so *3.* becomes a special instance of *1.*. If $i \neq j$ then by the definition of $(e_i^k]_j$ it follows that $e_j^l \prec e_i^k$. Since $e_j^{l'} \prec e_j^l$ for all $l' < l$ and since $\prec$ is transitive, it follows readily that $(e_j^l]_j \subseteq (e_i^k]_j$.

4. Since $(e_i^k]_j$ is a finite set of totally ordered events, it has a maximum element, say e_j^l. Hence, $(e_i^k]_j \subseteq (e_j^l]_j$. By *3.*, one also has $(e_j^l]_j \subseteq (e_i^k]_j$. □

Thus, by *4* above, one can uniquely and unambiguously encode a set $(e_i^k]_j$ by just a number, namely the size of the corresponding set $(e_j^l]_j$, i.e., the number of relevant events of thread t_j up to its l^{th} event. This suggests that if the DVC V_i maintained by $\mathcal{A}$ stores that number in its j^{th} component then *(a)* in Lemma 1 holds.

Before we formally show how reads and writes of shared variables affect the causal dependency relation, we need to introduce some notation. First, since a write of a shared variable introduces a causal dependency between the write event and all the previous read or write events of the same shared variable as well as all the events causally preceding those, we need a compact way to refer at any moment to all the read/write events of a shared variable, as well as the events that causally precede them. Second, since a read event introduces a causal dependency to all the previous write events of the same variable as well as all the events causally preceding those, we need a notation to refer to these events as well. Formally, if e_i^k is an event in a multithreaded computation $\mathcal{C}$ and $x \in S$ is a shared variable, then let

$$(e_i^k]_x^a = \begin{cases} \text{The thread-indexed set of all the relevant} \\ \text{events that are equal to or causally precede} \\ \text{an event } e \text{ } \textit{accessing } x\text{, such that } e \text{ occurs} \\ \text{before or it is equal to } e_i^k \text{ in } \mathcal{C}, \end{cases}$$

$$(e_i^k]_x^w = \begin{cases} \text{The thread-indexed set of all the relevant} \\ \text{events that are equal to or causally precede} \\ \text{an event } e \text{ } \textit{writing } x\text{, such that } e \text{ occurs} \\ \text{before or it is equal to } e_i^k \text{ in } \mathcal{C}. \end{cases}$$

It is obvious that $(e_i^k]_x^w \subseteq (e_i^k]_x^a$. Some or all of the thread-indexed sets of events above may be empty. By convention, if an event, say e, does not exist in $\mathcal{C}$, then we assume that the indexed sets $(e]$, $(e]_x^a$, and $(e]_x^w$ are all empty (rather than "undefined"). Note that if V_x^a and V_x^w in $\mathcal{A}$ store the corresponding numbers of elements in the index sets of $(e_i^k]_x^a$ and $(e_i^k]_x^w$ immediately after event e_i^k is processed by thread t_i, respectively, *(b)* and *(c)* in Lemma 1 hold.

Even though the sets of events $(e_i^k]$, $(e_i^k]_x^a$ and $(e_i^k]_x^w$ have mathematically clean definitions, they are based on total knowledge of the multithreaded computation $\mathcal{C}$. Unfortunately, $\mathcal{C}$ can be very large in practice, so the computation of these sets may be inefficient if not done properly. Since our analysis algorithms are *online*, we would like to calculate these sets *incrementally*, as the observer receives new events from the instrumented program. A key factor in devising efficient update algorithms is to find equivalent *recursive* definitions of these sets, telling us how to calculate a new set of events from similar sets that have been already calculated at previous event updates.

Let $\{e_i^k\}_i^{\mathcal{R}}$ be the indexed set whose j components are empty for all $j \neq i$ and whose i^{th} component is either the one element set $\{e_i^k\}$ when $e_i^k \in \mathcal{R}$ or the empty set otherwise. With the notation introduced, the following important recursive properties hold:

Lemma 3. *Let e_i^k be an event in $\mathcal{C}$ and let e_j^l be the event preceding*[1] *it in $\mathcal{C}$. If e_i^k is*

1. *An internal event then*
   $(e_i^k] \ = (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}}$,
   $(e_i^k]_x^a = (e_j^l]_x^a$, *for any* $x \in S$,
   $(e_i^k]_x^w = (e_j^l]_x^w$, *for any* $x \in S$;
2. *A read of x event then*
   $(e_i^k] \ = (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (e_j^l]_x^w$,
   $(e_i^k]_x^a = (e_i^k] \cup (e_j^l]_x^a$,
   $(e_i^k]_y^a = (e_j^l]_y^a$, *for any* $y \in S$ *with* $y \neq x$,
   $(e_i^k]_z^w = (e_j^l]_z^w$, *for any* $z \in S$;
3. *A write of x event then*
   $(e_i^k] \ = (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (e_j^l]_x^a$,
   $(e_i^k]_x^a = (e_i^k]$,
   $(e_i^k]_x^w = (e_i^k]$,
   $(e_i^k]_y^a = (e_j^l]_y^a$, *for any* $y \in S$ *with* $y \neq x$,
   $(e_i^k]_y^w = (e_j^l]_y^w$, *for any* $y \in S$ *with* $y \neq x$.

[1] If e_i^k is the first event then we can assume that e_j^l does not exist in $\mathcal{C}$, so by convention all the associated sets of events are empty

Proof: *1.* For the first equality, first recall that $e_i^k \in (e_i^k]$ if and only if e_i^k is relevant. Therefore, it suffices to show that $e \prec e_i^k$ if and only if $e \prec e_i^{k-1}$ for any relevant event $e \in \mathcal{R}$. Since e_i^k is internal, it cannot be in relation $<_x$ with any other event for any shared variable $x \in S$, so by the definition of $\prec$, the only possibilities are that either e is some event $e_i^{k'}$ of thread t_i with $k' < k$, or otherwise there is such an event $e_i^{k'}$ of thread t_i with $k' < k$ such that $e \prec e_i^{k'}$. Hence, it is either the case that e is e_i^{k-1} (so e_i^{k-1} is also relevant) or otherwise $e \prec e_i^{k-1}$. In any of these cases, $e \in (e_i^{k-1}]$. The other two equalities are straightforward consequences of the definitions of $(e_i^k]_x^a$ and $(e_i^k]_x^w$.

2. Like in the proof of *1.*, $e_i^k \in (e_i^k]$ if and only if $e_i^k \in \mathcal{R}$, so it suffices to show that for any relevant event $e \in \mathcal{R}$, $e \prec e_i^k$ if and only if $e \in (e_i^{k-1}] \cup (e_j^l]_x^w$. Since e_i^k is a read of $x \in S$ event, by the definition of $\prec$ one of the following must hold:

- $e = e_i^{k-1}$. In this case e_i^{k-1} is also relevant, so $e \in (e_i^{k-1}]$;
- $e \prec e_i^{k-1}$. It is obvious in this case that $e \in (e_i^{k-1}]$;
- e is a write of x event and $e <_x e_i^k$. In this case $e \in (e_j^l]_x^w$;
- There is some write of x event e' such that $e \prec e'$ and $e' <_x e_i^k$. In this case $e \in (e_j^l]_x^w$, too.

Therefore, $e \in (e_i^{k-1}]$ or $e \in (e_j^l]_x^a$.

Let us now prove the second equality. By the definition of $(e_i^k]_x^a$, one has that $e \in (e_i^k]_x^a$ if and only if e is equal to or causally precedes an event accessing $x \in S$ that occurs before or is equal to e_i^k in $\mathcal{C}$. Since e_i^k is a read of x, the above is equivalent to saying that either it is the case that e is equal to or causally precedes e_i^k, or it is the case that e is equal to or causally precedes an event accessing x that occurs *strictly before* e_i^k in $\mathcal{C}$. Formally, the above is equivalent to saying that either $e \in (e_i^k]$ or $e \in (e_j^l]_x^a$. If $y, z \in S$ and $y \neq x$ then one can readily see (like in *1.* above) that $(e_i^k]_y^a = (e_j^l]_y^a$ and $(e_i^k]_z^a = (e_j^l]_z^a$.

3. It suffices to show that for any relevant event $e \in \mathcal{R}$, $e \prec e_i^k$ if and only if $e \in (e_i^{k-1}] \cup (e_j^l]_x^a$. Since e_i^k is a write of $x \in S$ event, by the definition of $\prec$ one of the following must hold:

- $e = e_i^{k-1}$. In this case $e_i^{k-1} \in \mathcal{R}$, so $e \in (e_i^{k-1}]$;
- $e \prec e_i^{k-1}$. It is obvious in this case that $e \in (e_i^{k-1}]$;
- e is an access of x event (read or write) and $e <_x e_i^k$. In this case $e \in (e_j^l]_x^a$;
- There is some access of x event e' such that $e \prec e'$ and $e' <_x e_i^k$. In this case $e \in (e_j^l]_x^a$, too.

Therefore, $e \in (e_i^{k-1}]$ or $e \in (e_j^l]_x^a$.

For the second equality, note that, as for the second equation in *2.*, one can readily see that $e \in (e_i^k]_x^a$ if and only if $e \in (e_i^k] \cup (e_j^l]_x^a$. But $(e_j^l]_x^a \subseteq (e_i^k]$, so the above is equivalent to $e \in (e_i^k]$. A similar reasoning leads to $(e_i^k]_x^w = (e_i^k]$. The equalities for $y \neq x$ immediate, because e_i^k has no relation to accesses of other shared variables but x. □

Since each component set of each of the indexed sets in these recurrences has the form $(e_i^k]_i$ for appropriate i and k, and since each $(e_i^k]_i$ can be safely encoded by its size, one can then safely encode each of the above indexed sets by an n-dimensional DVC; these DVCs are precisely V_i for all $1 \leq i \leq n$ and V_x^a and V_x^w for all $x \in S$. Therefore, (a), (b) and (c) of Lemma 1 holds. An interesting observation is that one can regard the problem of recursively calculating $(e_i^k]$ as a dynamic programming problem. As can often be done in dynamic programming problems, one can reuse space and derive the Algorithm $\mathcal{A}$. The following theorem states that the DVC algorithm correctly implements causality in multithreaded programs.

Theorem 1. *If $\langle e, i, V\rangle$ and $\langle e', i', V'\rangle$ are two messages sent by $\mathcal{A}$, then $e \lhd e'$ if and only if $V[i] \leq V'[i]$ (no typo: the second i is not an i') if and only if $V < V'$.*

Proof: First, note that e and e' are both relevant. The case $i = i'$ is trivial. Suppose $i \neq i'$. Since, by *(a)* of Lemma 1, $V[i]$ is the number of relevant events that t_i generated before and including e and since $V'[i]$ is the number of relevant events of t_i that causally precede e', it is clear that $V[i] \leq V'[i]$ if and only if $e \prec e'$. For the second part, if $e \lhd e'$ then $V \leq V'$ follows again by *(a)* of Lemma 1, because any event that causally precedes e also precedes e'. Since there are some indices i and i' such that e was generated by t_i and e' by $t_{i'}$, and since $e' \not\lhd e$, by the first part of the theorem it follows that $V'[i'] > V[i']$; therefore, $V < V'$. For the other implication, if $V < V'$ then $V[i] \leq V'[i]$, so the result follows by the first part of the theorem. □

4 Runtime Model Generation and Predictive Analysis

In this section we consider what happens at the observer's site. The observer receives messages of the form $\langle e, i, V\rangle$. Because of Theorem 1, the observer can infer the causal dependency between the relevant events emitted by the multithreaded system. We show how the observer can be configured to effectively analyze all possible interleavings of events that do not violate the observed causal dependency *online* and *in parallel*. Only one of these interleavings corresponds to the real execution, the others being all potential executions. Hence, the presented technique can *predict* safety violations from successful executions.

4.1 Multithreaded Computation Lattice

Inspired by related definitions in [1], 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 major 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 computation, let $\mathcal{R}$ be the set of relevant events and $\lhd$ be the $\mathcal{R}$-relevant causality associated with the computation.

Definition 2 (Consistent Run). For a given permutation of events in $\mathcal{R}$, say $R = e_1 e_2 \ldots e_{|\mathcal{R}|}$, we say that R is a *consistent run* if for all pairs of events e and e', $e \lhd e'$ implies that e appears before e' in R.

Let e_i^k be the k^{th} relevant event generated by the thread t_i since the start of its execution. A *cut* C is a subset of $\mathcal{R}$ such that for all $i \in [1, n]$ if e_i^k is present in C then for all $l < k$, e_i^l is also present in C. A cut is denoted by a tuple $(e_1^{k_1}, e_2^{k_2}, ..., e_n^{k_n})$ where each entry corresponds to the last relevant event for each thread included in C. If a thread i has not seen a relevant event then the corresponding entry is denoted by e_i^0. A cut C corresponds to a relevant program state that has been reached after all the events in C have been executed. Such a relevant program state is called a *relevant global multithreaded state*, or simply a *relevant global state* or even just *state*, and is denoted by $\Sigma^{k_1 k_2 \ldots k_n}$.

Definition 3 (Consistent Cut). A cut is said to be consistent if for all events e and e'

$$(e \in C) \wedge (e' \lhd e) \rightarrow (e' \in C)$$

A consistent global state is the one that corresponds to a consistent cut. A relevant event e_i^l is said to be enabled in a consistent global state $\Sigma^{k_1k_2\ldots k_n}$ if and only if $C \cup \{e_i^l\}$ is a consistent cut, where C is the consistent cut corresponding to the state $\Sigma^{k_1k_2\ldots k_n}$. The following proposition holds for an enabled event:

Proposition 1. *A relevant event e_i^l is enabled in a consistent global state $\Sigma^{k_1k_2\ldots k_n}$ if and only if $l = k_i+1$ and for all relevant events e, if $e \neq e_i^l$ and $e \lhd e_i^l$ then $e \in C$, where C is the consistent cut corresponding to the state $\Sigma^{k_1k_2\ldots k_n}$.*

Proof. Since e_i^l is enabled in the state $\Sigma^{k_1k_2\ldots k_n}$, $C\cup\{e_i^l\}$ is a cut. This implies that for all events e_i^k, if $k < l$ then $e_i^k \in C \cup \{e_i^l\}$ and hence $e_i^k \in C$. In particular, all the events $e_i^1, e_i^2, \ldots, e_i^{l-1}$ are in C. However, e_i^{l-1} is the last relevant event from thread t_i which is included in C. Therefore, $k_i = l - 1$. Since $e_i^l \in C \cup \{e_i^l\}$, $e \lhd e_i^l$, and $C \cup \{e_i^l\}$ is a consistent cut, $e \in C \cup \{e_i^l\}$ (by the definition of consistent cut). Since by assumption $e \neq e_i^l$, we have $e \in C$. □

An immediate consequence of the above proposition is the following corollary:

Corollary 1. *If C is the consistent cut corresponding to the state $\Sigma^{k_1k_2\ldots k_n}$ and if e_i^l is enabled in $\Sigma^{k_1k_2\ldots k_n}$ then the state corresponding to the consistent cut $C \cup \{e_i^l\}$ is $\Sigma^{k_1k_2\ldots k_{i-1}lk_{i+1}\ldots k_n}$ or $\Sigma^{k_1k_2\ldots k_{i-1}(k_i+1)k_{i+1}\ldots k_n}$ and we denote it by $\delta(\Sigma^{k_1k_2\ldots k_n}, e_i^l)$.*

Here the partial function δ maps a consistent state Σ and a relevant event e enabled in that state to a consistent state $\delta(\Sigma, e)$ which is the result of executing e in Σ. Let Σ^{K_0} be the *initial* global state, $\Sigma^{00\ldots0}$, which is always consistent. The following result holds:

Lemma 4. *If $R = e_1e_2\ldots e_{|\mathcal{R}|}$ is a consistent multithreaded run then it generates a sequence of global states $\Sigma^{K_0}\Sigma^{K_1}\ldots\Sigma^{K_{|\mathcal{R}|}}$ such that for all $r \in [1, |\mathcal{R}|]$, $\Sigma^{K_{r-1}}$ is consistent, e_r is enabled in $\Sigma^{K_{r-1}}$, and $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$.*

Proof. The proof is by induction on r. By definition the initial state Σ^{K_0} is consistent. Moreover, e_1 is enabled in Σ^{K_0} because the cut C corresponding to the state Σ^{K_0} is empty and hence the cut $C \cup \{e_1\} = \{e_1\}$ is consistent. Since Σ^{K_0} is consistent and e_1 is enabled in Σ^{K_0}, $\delta(\Sigma^{K_0}, e_1)$ is defined. Let $\Sigma^{K_1} = \delta(\Sigma^{K_0}, e_1)$.

Let us assume that $\Sigma^{K_{r-1}}$ is consistent, e_r is enabled in $\Sigma^{K_{r-1}}$, and $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$. Therefore, $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$ is also consistent. Let C be the cut corresponding to Σ^{K_r}. To prove that e_{r+1} is enabled in Σ^{K_r} we have to prove that $C\cup e_{r+1}$ is a cut and it is consistent. Let $e_{r+1} = e_i^l$ for some i and l i.e. e_{r+1} is the l^{th} relevant event of thread t_i. For every event e_i^k such that $k < l$, $e_i^k \lhd e_i^l$. Therefore, by the definition of consistent run, e_i^k appears before e_i^l for all $0 < k < l$. This implies that all e_i^k for $0 < k < l$ are included in C. This proves that $C\cup e_i^l$ is a cut. Since C is a cut, for all events e and e' if $e \neq e_i^l$ then $(e \in C\cup\{e_i^l\}) \wedge (e' \lhd e) \rightarrow e' \in C\cup\{e_i^l\}$. Otherwise, if $e = e_i^l$ then by the definition of consistent run, if $e' \lhd e_i^l$ then e' appears before e_i^l in R. This implies that e' is included in $C\cup\{e_i^l\}$. Therefore, $C\cup\{e_i^l\}$ is consistent which proves that $e_{r+1} = e_i^l$ is enabled in the state Σ^{K_r}. Since, Σ^{K_r} is consistent and e_{r+1} is enabled in Σ^{K_r}, $\delta(\Sigma^{K_r}, e_{r+1})$ is defined. We let $\delta(\Sigma^{K_r}, e_{r+1}) = \Sigma^{K_{r+1}}$. □

From now on, we 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 $\Sigma \leadsto \Sigma'$, when there is some run in which Σ and Σ' are consecutive states. Let $\leadsto^*$ be the reflexive transitive closure of the relation $\leadsto$. The set of all consistent global states together with the relation $\leadsto^*$ forms a *lattice* with n mutually orthogonal axes representing each thread. For a state $\Sigma^{k_1k_2\ldots k_n}$, 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 $\Sigma^{r_1r_2\ldots r_n}$, where r_i is the total number of relevant events of thread t_i.

Therefore, a multithreaded computation can be seen as a lattice. This lattice, which is called *computation lattice* and referred to as $\mathcal{L}$, should be seen as an *abstract model* of the running multithreaded program, containing the relevant information needed in order to analyze the program. Supposing that one is able to *store* the computation lattice of a multithreaded program, which is a non-trivial matter because it can have an exponential number of states in the length of the execution, one can mechanically model-check it against the safety property.

Let $VC(e_i)$ be the DVC associated with the thread t_i when it generated the event e_i. Given a state $\Sigma^{k_1k_2\ldots k_n}$ we can associate a DVC with the state (denoted by $VC(\Sigma^{k_1k_2\ldots k_n})$) such that $VC(\Sigma^{k_1k_2\ldots k_n})[i] = k_i$ i.e. the i^{th} entry of $VC(\Sigma^{k_1k_2\ldots k_n})$ is equal to the number of relevant events of thread t_i that has causally effected the state. With this definition the following results hold:

Lemma 5. *If a relevant event e from thread t_i is enabled in a state Σ and if $\delta(\Sigma, e) = \Sigma'$ then $\forall j \neq i$: $VC(\Sigma)[j] = VC(\Sigma')[j]$ and $VC(\Sigma)[i] + 1 = VC(\Sigma')[i]$.*

Proof. This follows directly from the definition of DVC of a state and Corollary 1.

Lemma 6. *If a relevant event e from thread t_i is enabled in a state Σ then $\forall j \neq i$: $VC(\Sigma)[j] \geq VC(e)[j]$ and $VC(\Sigma)[i] + 1 = VC(e)[i]$.*

Proof. $VC(\Sigma)[i] + 1 = VC(e)[i]$ follows from Lemma 5. Say $k = VC(e)[j]$ for some $j \neq i$. Then by (a) of Lemma 1 we know that the k^{th} relevant event from thread t_j causally precedes e i.e. $e_j^k \lhd e$. Then by proposition 1 $e_j^k \in C$, where C is the cut corresponding to Σ. This implies that $k \leq VC(\Sigma)[j]$ which proves that $\forall j \neq i\colon VC(\Sigma)[j] \geq VC(e)[j]$.

Lemma 7. *If $R = e_1 e_2 \ldots e_{|\mathcal{R}|}$ is a consistent multithreaded run generating the sequence of global states $\Sigma^{K_0}\Sigma^{K_1}\ldots\Sigma^{K_{|\mathcal{R}|}}$, then $VC(\Sigma^{K_i})$ can be recursively defined as follows:*

$$
\begin{aligned}
&VC(\Sigma^{K_0})[j] = 0 \qquad \textit{for all } j \in [1, n] \\
&VC(\Sigma^{K_r})[j] = \mathit{max}\ (VC(\Sigma^{K_{r-1}})[j],\, VC(e_r)[j]) \\
&\qquad\qquad\qquad \textit{for all } j \in [1, n] \textit{ and } 0 < r \leq |\mathcal{R}|
\end{aligned}
$$

Proof. $\forall j \in [1, n]\colon VC(\Sigma^{K_0})[j] = 0$ holds by definition. Let e_r be from thread t_i. By Lemma 4 e_r is enabled in $\Sigma^{K_{r-1}}$. Therefore, by Lemma 6, $\forall j \neq i\colon VC(\Sigma^{K_{r-1}})[j] \geq VC(e_r)[j]$. This implies that $\forall j \neq i\colon VC(\Sigma^{K_r})[j] = VC(\Sigma^{K_{r-1}})[j] = \mathit{max}(VC(\Sigma^{K_{r-1}})[j], VC(e_r)[j])$. Otherwise if $j = i$, by Lemma 6, $VC(\Sigma^{K_{r-1}})[j] + 1 = VC(e_r)[j]$. Therefore, $VC(\Sigma^{K_r})[j] = VC(\Sigma^{K_{r-1}})[j] + 1 = VC(e_r)[j] = \mathit{max}(VC(\Sigma^{K_{r-1}})[j], VC(e_r)[j])$. This proves that $\forall j\colon VC(\Sigma^{K_r})[j] = \mathit{max}(VC(\Sigma^{K_{r-1}})[j], VC(e_r)[j])$.

Corollary 2. *If $R = e_1 e_2 \ldots e_{|\mathcal{R}|}$ is a consistent multithreaded run generating the sequence of global states $\Sigma^{K_0}\Sigma^{K_1}\ldots\Sigma^{K_{|\mathcal{R}|}}$, then*

$$
\begin{aligned}
VC(\Sigma^{K_r})[i] = \mathit{max}\ (VC(e_1)[i], VC(e_2)[i], \ldots, VC(e_r)[i]) \\
\textit{for all } i \in [1, n] \textit{ and } 0 < r \leq |\mathcal{R}|
\end{aligned}
$$

Example 2. Figure 2 shows the causal partial order on relevant events extracted by the observer from the multithreaded execution in Example 1, together with the generated computation lattice. The actual execution, $\Sigma^{00}\Sigma^{01}\Sigma^{11}\Sigma^{12}\Sigma^{22}\Sigma^{23}\Sigma^{33}$, is marked with solid edges in the lattice. Besides its DVC, each global state in the lattice stores its values for the relevant variables, w and v. It can be readily seen on Fig. 2 that the LTL property F_1 defined in Example 1 holds on the sample run of the system, and also that it is not in the language of bad prefixes, F_2. However, F_1 is violated on some other consistent runs, such as $\Sigma^{00}\Sigma^{01}\Sigma^{02}\Sigma^{12}\Sigma^{13}\Sigma^{23}\Sigma^{33}$. On this particular run $\uparrow p$ holds at Σ^{02}; however, r does not hold at the next state Σ^{12}. This makes the formula F_1 false at the state Σ^{13}. The run can also be symbolically written as $\{\}\{\}\{p\}\{p\}\{p,q\}\{p,q,r\}\{p,q,r\}$. In the automaton in Fig. 1, this corresponds to a possible sequence of states 00123555. Hence, this string is accepted by F_2 as a bad prefix.

Therefore, by carefully analyzing the computation lattice extracted from a successful execution one can infer safety violations in other possible consistent executions. Such violations give informative feedback to users, such as the lack of synchronization in the example above, and may be hard to find by just ordinary testing. In what follows we propose effective techniques to analyze the computation lattice. A first important observation is that one can generate it *on-the-fly* and analyze it on a level-by-level basis, discarding the previous levels. However, even if one considers only one level, that can still contain an exponential number of states in the length of the current execution. A second important observation is that the states in the computation lattice are not all equiprobable in practice. By allowing a user configurable *window* of most likely states in the lattice centered around the observed execution trace, the presented technique becomes quite scalable, requiring $O(wm)$ space and $O(twm)$ time, where w is the size of the window, m is the size of the bad prefix monitor of the safety property, and t is the size of the monitored execution trace.

4.2 Level By Level Analysis of the Computation Lattice

A naive observer of an execution trace of a multithreaded program would just check the observed execution trace against the monitor for the safety property, say $\mathcal{M}on$ like in Definition 1, and would maintain at each moment a set of states, say *MonStates* in $\mathcal{M}$. When a new event arrives, it would create the next state Σ and replace *MonStates* by $\rho(\mathit{MonStates}, \Sigma)$. If the bad state b will ever be in *MonStates* then a property violation error would be reported, meaning that the current execution trace led to a bad prefix of the safety property. Here we assume that the events are received in the order in which they are emitted, and also that the monitor works over the global states of the multithreaded programs. This assumption is essential for the observer to deduce the actual execution of the multithreaded program. The knowledge of the actual execution is used by the observer to apply the causal cone heuristics as described later. The assumption is not necessary if we do not want to use causal cone heuristics. The work in [22] describes a technique for the level by level analysis of the computation lattice without the above assumption.

A smart observer, as said before, will analyze not only the observed execution trace, but also all the other consistent runs of the multithreaded system, thus being able to *predict* violations from successful executions. The observer receives the events from the running multithreaded program in real-time and enqueues them in an event queue Q. At the same time, it traverses the computation lattice level by level and checks whether the bad state of the monitor can be hit by any of the runs up to the current level. We next provide the algorithm that the observer uses to construct the lattice level by level from the sequence of events it receives from the running program.

The observer maintains a list of global states (*CurrLevel*), that are present in the current level of the lattice. For each event e in the event queue, it tries to construct a new global state from the set of states in the current level and the event e. If the global state is created

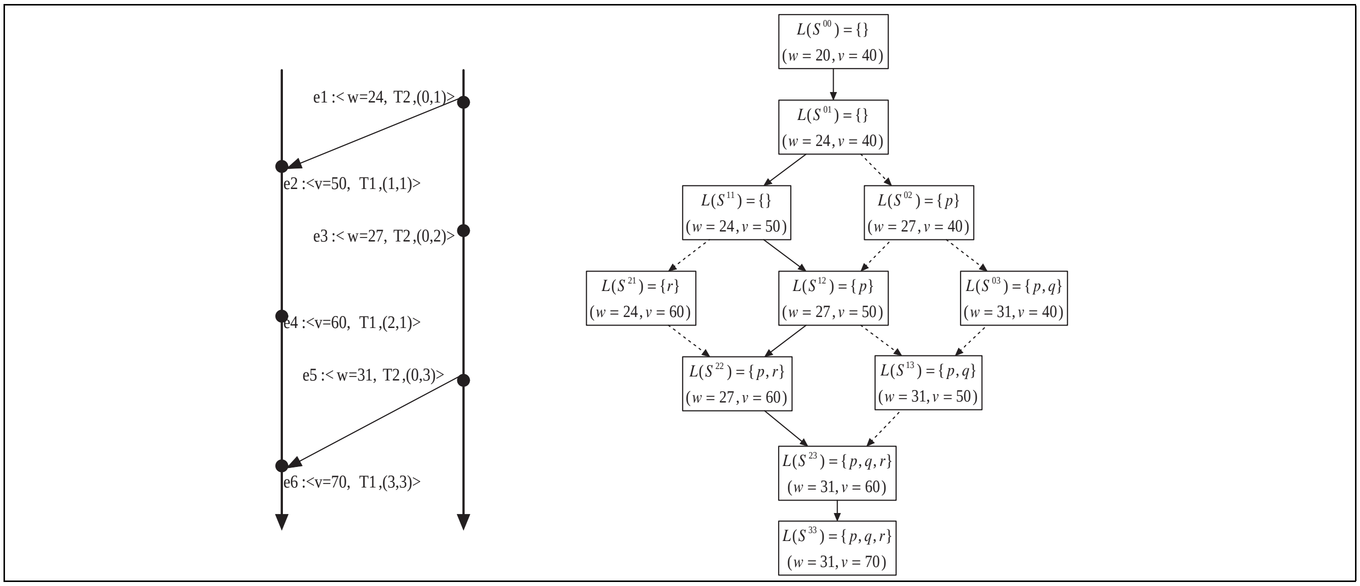

Fig. 2. Computation Lattice

successfully then it is added to the list of global states (*NextLevel*) for the next level of the lattice. The process continues until certain condition, *levelComplete?()* holds. At that time the observer says that the level is complete and starts constructing the next level by setting *CurrLevel* to *NextLevel*, *NextLevel* to empty set, and reallocating the space previously occupied by *CurrLevel*. Here the predicate *levelComplete?()* is crucial for generating only those states in the level that are most likely to occur in other executions, namely those in the *window*, or the *causality cone*, that is described in the next subsection. The *levelComplete?* predicate is also discussed and defined in the next subsection. The pseudo-code for the lattice traversal is given in Fig. 3.

Every global state Σ contains the value of all relevant shared variables in the program, a DVC $VC(\Sigma)$ to represent the latest events from each thread that resulted in that global state. Here the predicate *nextState?*(Σ, e), checks if the event e is enabled in the state Σ, where *threadId(e)* returns the index of the thread that generated the event e, $VC(\Sigma)$ returns the DVC of the global state Σ, and *VC(e)* returns the DVC of the event e. The correctness of the function is given by Lemma 6. It essentially says that event e can generate a consecutive state for a state Σ, if and only if Σ "knows" everything e knows about the current evolution of the multithreaded system except for the event e itself. Note that e may know less than Σ knows with respect to the evolution of other threads in the system, because Σ has global information.

The function *createState*(Σ, e), which implements the function δ described in Corollary 1 creates a new global state Σ', where Σ' is a possible consistent global state that can result from Σ after the event e. Together with each state Σ in the lattice, a set of states of the

```
while(not end of computation){
    Q ← enqueue(Q, NextEvent())
    while(constructLevel()){}
  }

boolean constructLevel(){
  for each e ∈ Q and Σ ∈ CurrLevel {
    if nextState?(Σ, e) {
      NextLevel ← NextLevel ⊎ createState(Σ, e)
      if levelComplete?(NextLevel, e, Q) {
        Q ← removeUselessEvents(CurrLevel, Q)
        CurrLevel ← NextLevel
        NextLevel ← ∅
        return true}}}
  return false
}

boolean nextState?(Σ, e){
  i ← threadId(e);
  if (∀j ≠ i : VC(Σ)[j] ≥ VC(e)[j] and
        VC(Σ)[i] + 1 = VC(e)[i]) return true
  return false
}

State createState(Σ, e){
  Σ' ← new copy of Σ
  j ← threadId(e);  VC(Σ')[j] ← VC(Σ)[j] + 1
  pgmState(Σ')[var(e) ← value(e)]
  MonStates(Σ') ← ρ(MonStates(Σ), Σ')
  if b ∈ MonStates(Σ') {
    output 'property may be violated'}
  return Σ'
}
```

Fig. 3. Level-by-level traversal.

monitor, *MonStates(*Σ*)*, also needs to be maintained, which keeps all the states of the monitor in which any of the partial runs ending in Σ can lead to. In the function *createState*, we set the *MonStates* of Σ' with the set of monitor states to which any of the current states in *MonStates*(Σ) can transit when the state Σ' is observed. *pgmState*(Σ') returns the value of all relevant program shared variables in state Σ', *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 *pgmState*(Σ')[*var*(e) $\leftarrow$ *value*(e)] means that in *pgmState*(Σ'), *var(e)* is updated with *value(e)*. Lemma 5 justifies that DVC of the state Σ' is updated properly.

The merging operation *nextLevel*$\uplus\,\Sigma$ adds the global state Σ to the set *nextLevel*. If Σ is already present in *nextLevel*, it updates the existing state's *MonStates* with the union of the existing state's *MonStates* and the *Monstates* of Σ. Two global states are same if their DVCs are equal. Because of the function *levelComplete?*, it may be often the case that the analysis procedure moves from the current level to the next one before it is exhaustively explored. That means that several events in the queue, which were waiting for other events to arrive in order to generate new states in the current level, become unnecessary so they can be discarded. The function *removeUselessEvents(CurrLevel,Q)* removes from Q all the events that cannot contribute to the construction of any state at the next level. It creates a DVC V_{min} whose each component is the minimum of the corresponding component of the DVCs of all the global states in the set *CurrLevel*. It then removes all the events in Q whose DVCs are less than or equal to V_{min}. This function makes sure that we do not store any unnecessary events. The correctness of the function is given by the following lemma.

Lemma 8. *For a given relevant event e, if* $VC(e) \leq V_{min}$ *then* $\forall\Sigma \in$ *CurrLevel, e is not enabled in* Σ*.*

Proof. If e is enabled in Σ then by Lemma 6, $VC(e)[i] = VC(\Sigma)+1$, where t_i is the thread that generated e. This implies that if e is enabled in Σ then $VC(e) \not\leq VC(\Sigma)$. Since $VC(e) \leq V_{min}$ we have $\forall\Sigma \in CurrLevel$, $VC(e) \leq VC(\Sigma)$. Therefore, e is not enabled in Σ.

The observer runs in a loop till the computation ends. In the loop the observer waits for the next event from the running instrumented program and enqueues it in Q whenever it becomes available. After that the observer runs the function *constructLevel* in a loop till it returns false. If the function *constructLevel* returns false then the observer knows that the level is not completed and it needs more events to complete the level. At that point the observer again starts waiting for the next event from the running program and continues with the loop. The pseudo-code for the observer is given at the top of Fig. 3.

4.3 Causality Cone Heuristic

The number of states on a level in the computation lattice can be exponential in the length of the trace. In online analysis, generating all the states in a level may not be feasible. However, note that some states in a level can be considered more likely to occur in a consistent run than others. For example, two independent events that can possibly permute may have a huge time difference. Permuting these two events would give a consistent run, but that run may not be likely to take place in a real execution of the multithreaded program. So we can ignore such a permutation. We formalize this concept as *causality cone*, or *window*, and exploit it in restricting our attention to a small set of states in a given level.

As mentioned earlier we assume that the events are received in an order in which they happen in the computation. We can ensure this linear ordering by executing the DVC algorithm in a synchronized block so that each such execution takes place atomically with respect to each other. Note that this ordering gives the real execution of the program and it respects the partial order associated with the computation. This execution will be taken as a reference in order to compute the most probable consistent runs of the system.

If we consider all the events generated by the executing distributed program as a finite sequence of events, then a lattice formed by any prefix of this sequence is a sub-lattice of the computation lattice $\mathcal{L}$. This sub-lattice, say $\mathcal{L}'$, has the following property: if $\Sigma \in \mathcal{L}'$, then for any $\Sigma' \in \mathcal{L}$ if $\Sigma' \leadsto^* \Sigma$ then $\Sigma' \in \mathcal{L}'$. We can see this sub-lattice as a portion of the computation lattice $\mathcal{L}$ enclosed by a cone. The height of this cone is determined by the length of the current sequence of events. We call this *causality cone*. All the states in $\mathcal{L}$ that are outside this cone cannot be determined from the current sequence of events. Hence, they are outside the causal scope of the current sequence of events. As we get more events this cone moves down by one level.

If we compute a DVC V_{max} whose each component is the maximum of the corresponding component of the DVCs of all the events in the event queue V_{max} represents the DVC of the global state appearing at the tip of the cone. The tip of the cone, by Corollary 2, traverses the actual execution run of the program.

To avoid the generation of a possibly exponential number of states in a given level, we consider a fixed number, say w, of most probable states in a given level. In a level construction, we say that the level is complete once we have generated w states in that level. However, a level may contain less than w states. Then the level construction algorithm gets stuck. Moreover, we cannot determine if a level has less than w states unless we see all the events in the complete computation. This is because we do not know the total number of threads that participate in the computation beforehand. To avoid this scenario we introduce another parameter l, the length of

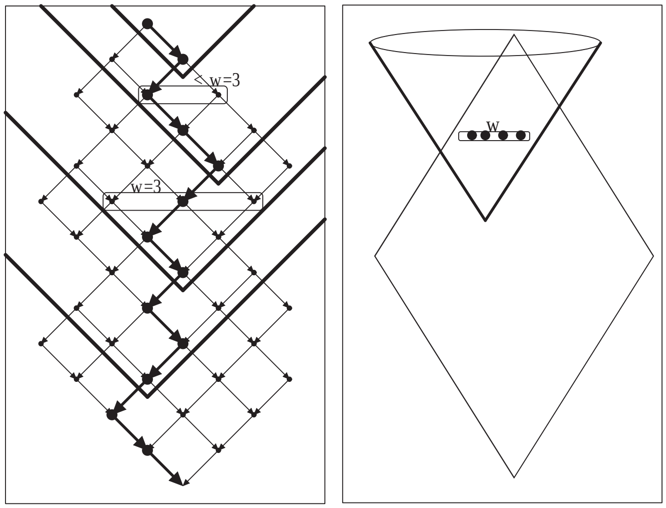

Fig. 4. Causality Cones

```
boolean levelComplete?(NextLevel, e, Q){
  if size(NextLevel) ≥ w then
      return true;
  else if e is the last event in Q
    and size(Q) == l then
      return true;
  else return false;
}
```

Fig. 5. *levelComplete?* predicate

the current event queue. We say that a level is complete if we have used all the events in the event queue for the construction of the states in the current level and the length of the queue is l and we have not crossed the limit w on the number of states. The pseudo-code for *levelComplete?* is given in Fig. 5

Note, here l corresponds to the number of levels of the sub-lattice that can be constructed from the events in the event queue Q. This is because, by Corollary 1, every event in the queue Q generates a state in the next level from a state in the current level in which it is enabled.

Example 3. Figure 6 shows the portion of the computation lattice constructed from the multithreaded execution in Example 1 when the causality cone heuristics is applied with parameters $w = 2$ and $l = 3$. The possible consistent run $\Sigma^{00}\Sigma^{01}\Sigma^{02}\Sigma^{03}\Sigma^{13}\Sigma^{23}\Sigma^{33}$, shown on the left side of the Figure 6, is pruned out by the heuristics. In this particular run the two independent events $e2$ and $e5$ that are permuted have long time difference in the actual execution. Therefore, we can safely ignore this run among all other possible consistent runs.

5 Implementation

We have implemented these new techniques, in version 2.0 of the tool Java MultiPathExplorer (JMPaX), which has been designed to monitor multithreaded Java programs. The current implementation is written in Java. The tool has three main modules, the *instrumentation* module, the *observer* module and the *monitor* module.

The instrumentation program, named `instrument`, takes a specification file and a list of class files as command line arguments. An example is

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

where the specification file `spec` contains a list of named formulae written in a suitable logic. The program `instrument` extracts the name of the relevant variables from the specification and instruments the classes, provided in the argument, as follows:

i) For each variable `x` of primitive type in each class it adds *access* and *write* DVCs, namely `_access_dvc_x` and `_write_dvc_x`, as new fields in the class.
ii) It adds code to associate a DVC with every newly created thread;
iii) For each read and write access of a variable of primitive type in any class, it adds codes to update the DVCs according to the algorithm mentioned in Section 3.4;
iv) It adds code to call a method `handleEvent` of the *observer* module at every write of a relevant variable.

The instrumentation module uses the BCEL [5] Java library to modify Java class files. Currently, the instrumentation module instruments every variable of primitive type in every class. This implies that, during the execution of an instrumented program, the DVC algorithm is executed for every read and write of variables of primitive type. This degrades the performance of the program considerably. We plan to improve the instrumentation by using *escape analysis* [4], to detect the possible shared variables through static analysis. This will reduce the number of instrumentation points and hence will improve the performance.

The *observer* module, that takes two parameters w and l, generates the lattice level-by-level when the instrumented program is executed. Whenever the `handleEvent` method is invoked, it enqueues the event passed as argument to the method `handleEvent`. Based on the event queue and the current level of the lattice, it generates the next level. In the process, it invokes the `nextStates` method (corresponding to ρ in a *monitor*) of the *monitor* module.

The *monitor* module reads the specification file written either as an LTL formula or as a regular expression and generates the non-deterministic automaton corresponding to the formula or the regular expression. It provides the method `nextStates` as an interface to the *observer* module. The method raises an exception if at any point the set of states returned by `nextStates` contains the "bad" state of the automaton. The system be-

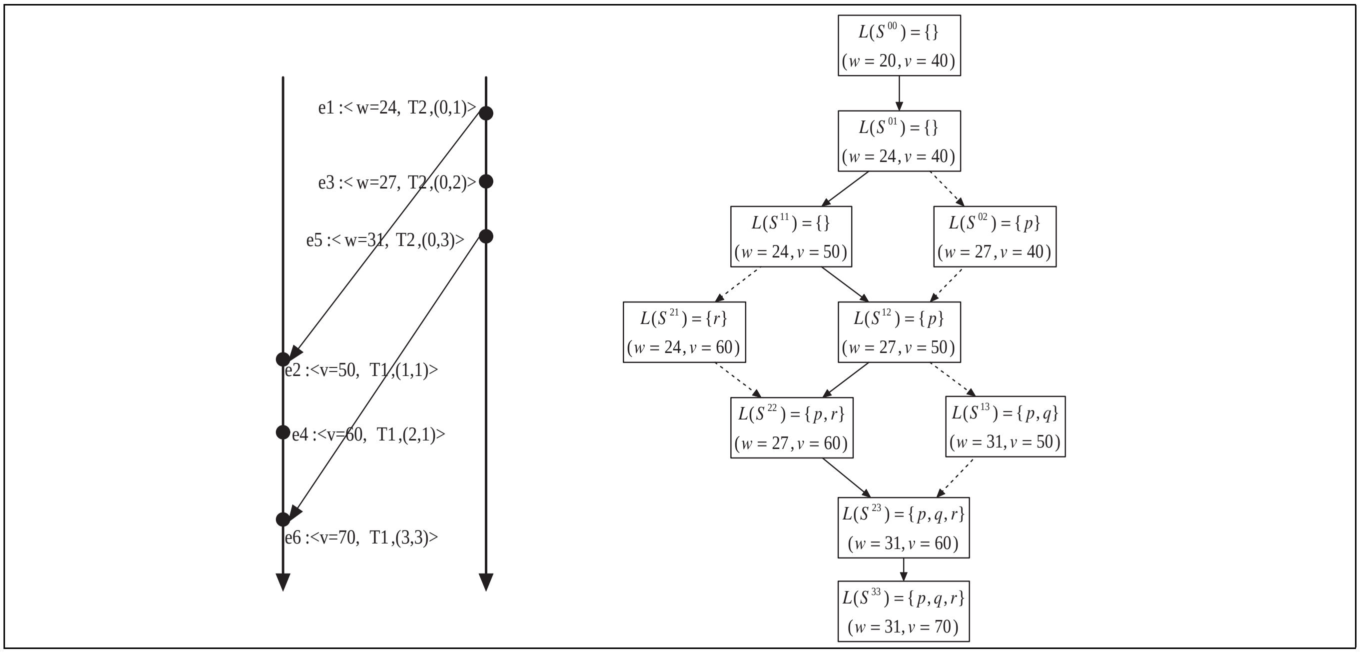

Fig. 6. Causality Cone Heuristics applied to Example 2

ing modular, the user can plug in his/her own *monitor* module for his/her logic of choice.

Since in Java synchronized blocks cannot be interleaved, so corresponding events cannot be permuted, locks are considered as shared variables and a write event is generated whenever a lock is acquired or released. This way, a causal dependency is generated between any exit and any entry of a synchronized block, namely the expected happens-before relation. Java synchronization statements are handled exactly the same way, that is, the shared variable associated to the synchronization object is written at the entrance and at the exit of the synchronized region. Condition synchronizations (wait/notify) can be handled similarly, by generating a write of a dummy shared variable by both the notifying thread before notification and by the notified thread after notification.

Note that the above technique for handing synchronization constructs is conservative as it prevents us from permuting two synchronized blocks even if the events in the two blocks are independent of each other. Future work involves finding an extension of the DVC algorithm that can allow such permutations. This will enable us to extract more interleavings from a computation.

6 Conclusion and Future Work

A formal runtime predictive analysis technique for multithreaded systems has been presented in this paper, in which multiple threads communicating by shared variables are automatically instrumented to send relevant events, stamped by dynamic vector clocks, to an external observer which extracts a causal partial order on the global state, updates and thereby builds an abstract runtime model of the running multithreaded system. Analyzing this model on a level-by-level basis, the observer can infer effectively, from a *successful execution* of the observed system, when safety properties can be violated by *other executions*. Attractive future work includes predictions of liveness violations and predictions of data-race and deadlock conditions.

7 Acknowledgments

The work is supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586, the DARPA IXO NEST Program, contract number F33615-01-C-1907), the ONR Grant N00014-02-1-0715, the Motorola Grant MOTOROLA RPS #23 ANT, and the joint NSF/NASA grant CCR-0234524.

References

1. O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.
2. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57, Venice, Italy, January 2004. Springer-Verlag.

3. 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.
4. J. D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '99)*, volume 34(10) of *SIGPLAN Notices*, pages 1–19, Denver, Colorado, USA, November 1999.
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. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of Formal Methods Europe (FME'93): Industrial Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284, 1993.
7. D. Drusinsky. Temporal rover. http://www.time-rover.com.
8. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
9. D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *Proc. of CAV'03: Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 114–118, Boulder, Colorado, USA, 2003. Springer-Verlag.
10. 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.
11. E. L. Gunter, R. P. Kurshan, and D. Peled. PET: An interactive software testing tool. In *Computer Aided Verification (CAV'00)*, volume 1885 of *Lecture Notes in Computer Science*, pages 552–556. Springer-Verlag, 2003.
12. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
13. K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
14. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
15. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
16. R. Lencevicius, A. Ran, and R. Yairi. Third eye - specification–based analysis of software execution traces. In *International Workshop on Automated Program Analysis, Testing and Verification (Workshop of ICSE 2000)*, pages 51–56, June 2000.
17. K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WADG'91)*, volume 579 of *Lecture Notes in Computer Science*, pages 254–272. Springer-Verlag, 1991.
18. F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1989.
19. A. Sen and V. K. .Garg. Partial order trace analyzer (pota) for distrubted programs. In *Proceedings of the 3rd Workshop on Runtime Verification (RV'03)*, Electronic Notes in Theoretical Computer Science, 2003.
20. K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. In *Proceedings of the 3rd Workshop on* Runtime Verification (RV'03), volume 89 of *ENTCS*, pages 162–181. Elsevier Science, 2003.
21. K. Sen, G. Roşu, and G. Agha. Generating Optimal Linear Temporal Logic Monitors by Coinduction. In *Proceedings of 8th Asian Computing Science Conference (ASIAN'03)*, volume 2896 of *Lecture Notes in Computer Science*, pages 260–275. Springer-Verlag, December 2003.
22. K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proceedings of 4th joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*. ACM, 2003.
23. K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 123–138, Barcelona, Spain, March 2004.
24. O. Shtrichman and R. Goldring. The 'logic-assurance' system - a tool for testing and controlling real-time systems. In *Proc. of the Eighth Israeli Conference on computer systems and sofware engineering (ICCSSE97)*, pages 47–55, June 1997.
25. L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). pages 1–9. ACM Press, 1973.
26. S. A. Vilkomir and J. P. Bowen. Formalization of software testing criteria using the Z notation. In *Proceedings of COMPSAC 01: 25th IEEE Annual International Computer Software and Applications Conference*, pages 351–356. IEEE Computer Society, Oct. 2001.