Online Efficient Predictive Safety Analysis of Multithreaded Programs

Koushik Sen, Grigore Roşu, and Gul Agha

Department of Computer Science,
University of Illinois at Urbana-Champaign.
{ksen,grosu,agha}@cs.uiuc.edu

Abstract. An automated and configurable technique for runtime safety analysis of multithreaded programs is presented, which is able to *predict* safety violations from successful executions. Based on a user provided safety formal specification, the program is automatically instrumented to emit *relevant* state update events to an observer, which further checks them 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 analyzed *on-line* and *in parallel*, and a warning is issued whenever there is a trace violating the specification. This technique can be therefore seen as a bridge between testing and model checking. To further increase scalability, a *window* in the state space can be specified, allowing the observer to infer the *most probable* runs. If the size of the window is 1 then only the received execution trace is analyzed, like in testing; if the size of the window is ∞ then all the execution traces are analyzed, such as in model checking.

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 heavy and lighter techniques to detect errors in multithreaded systems have been extensively investigated. The heavy techniques include traditional formal methods based approaches, such as model checking and theorem proving, guaranteeing that a formal model of the system satisfies its safety requirements by exploring, directly or indirectly, all possible thread interleavings. On the other hand, the lighter techniques include testing, that scales well and is one of the most used approaches to validate software products today.

As part of our overall effort in merging testing and formal methods, aiming at getting some of the benefits of both while avoiding the pitfalls of ad hoc testing and the complexity of full-blown model checking or theorem proving, in this paper we present a *runtime verification* technique for safety analysis of multithreaded systems, that can be tuned to analyze from one trace to all traces that are consistent with an actual execution of the program. If all traces are checked, then it becomes equivalent to online model checking of an abstract model of the computation, called the *multithreaded computation lattice*, which is extracted from the actual execution trace of the program, like in POTA [10] or JMPaX [14]. If only one trace is considered, then our technique becomes equivalent to checking just the actual execution of the multithreaded program, like in testing or like in other runtime analysis tools like MaC [7] and PaX [5, 1]. In general, depending on the application, one can configure a window within the state space to be explored, called *causality cone*, intuitively giving a causal "distance" from the observed execution within which all traces are exhaustively verified. 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 all *in parallel* at a cost which in the worst case is proportional with both the size of the window and the size of the state space of the monitor.

There are three important interrelated components of the proposed runtime verification technique namely *instrumentor*, *observer* and *monitor*. The code instrumentor, based on the safety specification, entirely automatically adds code to emit events when *relevant* state updates occur. 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. As layers are completed, the monitor, which is synthesized automatically from the safety specification, checks them against the safety specification and then discards them.

The concepts and notions presented in this paper have been experimented and tested on a practical monitoring system for Java programs, JMPaX 2.0, that extends its predecessor JMPaX [12] in at least four non-trivial novel ways. First, it introduces the technical notion of *dynamic vector clock*, allowing it to properly deal with dynamic creation and destruction of threads. Second, the variables shared between threads do not need to be static anymore: an automatic instrumentation technique has been devised that detects automatically when a variable is shared. Thirdly, and perhaps most importantly, the notion of *cone heuristic*, or *global state window*, is introduced for the first time in JMPaX 2.0 to increase the runtime efficiency by analyzing the most likely states in the computation lattice. Lastly, the presented runtime prediction paradigm is safety formalism independent, 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).

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 just any 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 has some easy to detect property, such as a specific variable having a negative value.

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, [11,13] show coinductive techniques to generate minimal static monitors from EREs and from future time linear temporal logics, respectively, and [6,1] 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.

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 for the implementation 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);
}
```

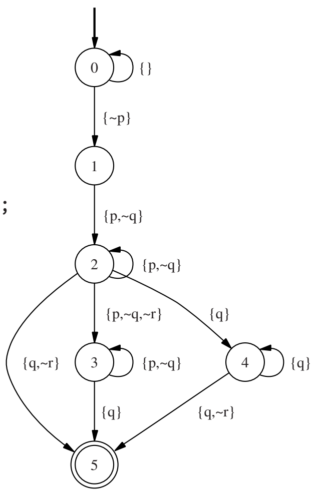

Fig. 1. Two threads (T1 controls the valve and T2 reads the water level) and a monitor.

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, 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:

$$F_1 = \Box(q \to ((r \wedge p)\mathcal{S} \uparrow p)) \tag{1}$$

$$F_2 = \{\}^*\{\neg p\}\{p, \neg q\}^+(\{p, \neg q, \neg r\}\{p, \neg q\}^*\{q\} + \{q\}^*\{q, \neg r\})\{\}^* \tag{2}$$

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 $e_1 e_2 \dots e_r$ 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 $\mathcal{M}$, 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 $\mathcal{M}$. 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 $\mathcal{M}$ 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 $\mathcal{M}$. 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 $e_1, e_2, \dots,$ 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 re-orderings 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 [6], this can be compactly written as $(x > 0) \rightarrow [y = 0, y > z)$. 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 [8] 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* [4,9] that correctly and efficiently implements the relevant causality relation. Let $V : \mathit{ThreadId} \rightarrow \mathit{Nat}$ 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 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.

The following theorem states that the DVC algorithm correctly implements causality in multithreaded programs. This algorithm has been previously presented by the authors in [14,15] in a less general context, where the number of threads is fixed and known a priori. Its proof is similar to that in [15].

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

- *$V_i[j]$ 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;*
- *$V_x^a[j]$ equals the number of relevant events of t_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;*
- *$V_x^w[j]$ 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.*

Therefore, if $\langle e, i, V \rangle$ and $\langle e', j, V' \rangle$ are two messages sent by dynamic vector clock algorithm, then $e \lhd e'$ if and only if $V[i] \leq V'[i]$. Moreover, if i and j are not given, then $e \lhd e'$ if and only if $V < V'$.

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 [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 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.

We let $\mathcal{R}$ denote the set of received relevant events. For a given permutation of events in $\mathcal{R}$, say $e_1 e_2 \ldots e_{|\mathcal{R}|}$, we let e_i^k denote the k-th event of thread t_i. 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 $\Sigma \leadsto \Sigma'$, 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 $\Sigma^{k_1 k_2 \ldots k_n}$, 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 t_i. Note that in the above discussion we assumed a fixed number of threads n. In a program where threads can be created and destroyed dynamically, only those threads are considered that at the end of the computation have causally affected the final values of the relevant variables.

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.

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 vari-

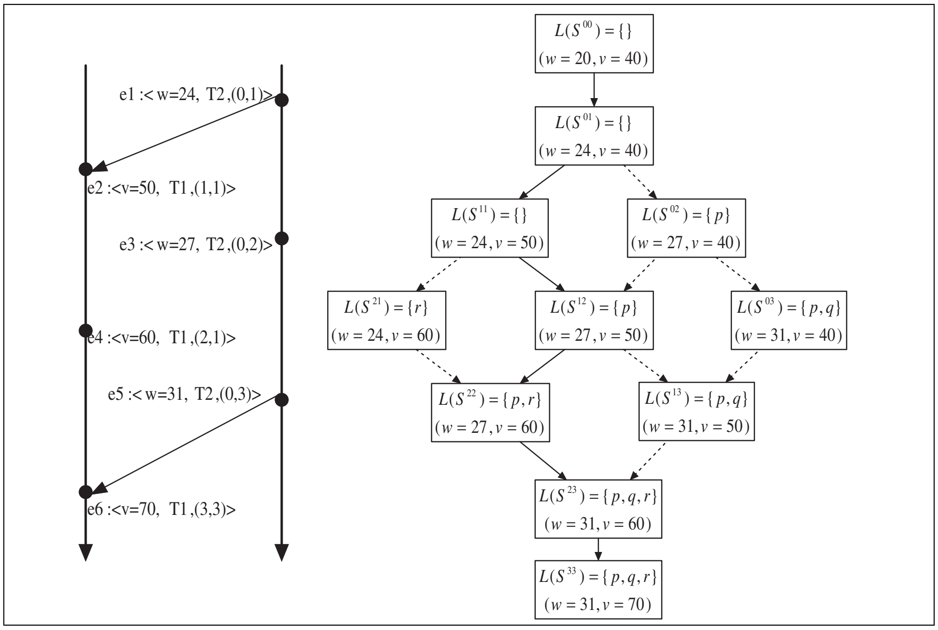
e1 :< w=24, T2,(0,1)>
e2 :<v=50, T1,(1,1)>
e3 :< w=27, T2,(0,2)>
e4 :<v=60, T1,(2,1)>
e5 :< w=31, T2,(0,3)>
e6 :<v=70, T1,(3,3)>
L(S^00) = {} (w = 20, v = 40)
L(S^01) = {} (w = 24, v = 40)
L(S^11) = {} (w = 24, v = 50)
L(S^02) = {p} (w = 27, v = 40)
L(S^21) = {r} (w = 24, v = 60)
L(S^12) = {p} (w = 27, v = 50)
L(S^03) = {p, q} (w = 31, v = 40)
L(S^22) = {p, r} (w = 27, v = 60)
L(S^13) = {p, q} (w = 31, v = 50)
L(S^23) = {p, q, r} (w = 31, v = 60)
L(S^33) = {p, q, r} (w = 31, v = 70)

Fig. 2. Computation Lattice

ables, 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 generating a new global state Σ arrives, it would 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.

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 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* 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 can convert the state Σ to a state Σ' in the next level of the lattice, 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. 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.

```
while(not end of computation){
  $Q \leftarrow enqueue(Q, NextEvent())$
  while($constructLevel()$){}
}

boolean $constructLevel()${
  for each $e \in Q$ {
    if $\Sigma \in CurrLevel$ and $nextState?(\Sigma, e)$ {
      $NextLevel \leftarrow NextLevel \uplus createState(\Sigma, e)$
      if $levelComplete?(NextLevel, e, Q)$ {
        $Q \leftarrow removeUselessEvents(CurrLevel, Q)$
        $CurrLevel \leftarrow NextLevel$
        return $true$}}}
  return $false$
}
boolean $nextState?(\Sigma, e)${
  $i \leftarrow threadId(e)$;
  if $(\forall j \neq i : VC(\Sigma)[j] \geq VC(e)[j]$ and
     $VC(\Sigma)[i] + 1 = VC(e)[i])$ return $true$
  return $false$
}
State $createState(\Sigma, e)${
  $\Sigma' \leftarrow$ new $copy\ of\ \Sigma$
  $j \leftarrow threadId(e);\ VC(\Sigma')[j] \leftarrow VC(\Sigma)[j] + 1$
  $pgmState(\Sigma')[var(e) \leftarrow value(e)]$
  $MonStates(\Sigma') \leftarrow \rho(MonStates(\Sigma), \Sigma')$
  if $b \in MonStates(\Sigma')$ {
    output 'property may be violated'}
  return $\Sigma'$
}
```

Fig. 3. Level-by-level traversal.

The function $createState(\Sigma, e)$ 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 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 $MonStates(\Sigma)$ can transit within the monitor when the state Σ' is observed. $pgmState(\Sigma')$ 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 $pgmState(\Sigma')[var(e) \leftarrow value(e)]$ means that in $pgmState(\Sigma')$, *var(e)* is updated with *value(e)*.

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. To do so, 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 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

In a given level of a computation lattice, the number of states can be large; in fact, 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.

In what follows we assume that the events are received in an order in which they happen in the computation. This is easily ensured by proper instrumentation. 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 sublattice of the computation lattice $\mathcal{L}$. This sublattice, 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 sublattice 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, then this represents the DVC of the global state appearing at the tip of the cone. The tip of the cone traverses the actual execution run of the program.

To avoid the generation of possibly exponential number of states in a given level, we consider a fixed number, say w, most probable states in a given level. In a level construction we say 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.

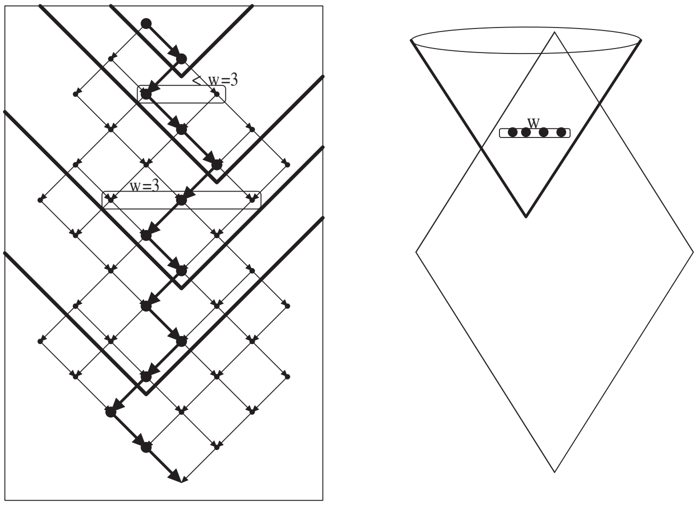

Fig. 4. Causality Cones

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 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 pseudocode for *levelComplete?* is given in Fig. 5.

```
boolean $levelComplete?(NextLevel, e, Q)${
 if $size(NextLevel) \geq w$ 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

Note, here l corresponds to the number of levels of the sublattice that be constructed from the events in the event queue Q. On the other hand, the level of this sublattice with the largest level number and having at least w global states refers to the $CurrLevel$ in the algorithm.

5 Implementation

We have implemented these new techniques, in version 2.0 of the tool Java MultiPathExplorer (JMPaX)[12], which has been designed to monitor multithreaded Java programs. The current implementation is written in Java and it removes the restriction that all the shared variables of the multithreaded program are static variables of type `int`. 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 BCEL [3] Java library to modify Java class files. We use the BCEL library to get a better handle for a Java classfile.

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 of the lattice. In the process it invokes `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 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` contain the "bad" state of the automaton. The system being modular, 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.

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 *successful* execution of the observed system when basic safety properties can be violated by other executions. Attractive future work includes predictions of liveness violations and predictions of datarace and deadlock conditions.

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. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of Fifth International VMCAI conference (VMCAI'04) (To appear in LNCS)*, January 2004.
   Download: http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp24.pdf.
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. M. Dahm. Byte code engineering with the bcel api. Technical Report B-17-98, Freie Universit at Berlin, Institut für Informatik, April 2001.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.

11. 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.
12. K. Sen, G. Roşu, and G. Agha. Java MultiPathExplorer (JMPaX 2.0). Download: `http://fsl.cs.uiuc.edu/jmpax`.
13. 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.
14. 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.
15. 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.