Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions

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

Abstract. A predictive runtime analysis technique is proposed for detecting violations of safety properties from apparently successful executions of concurrent systems. In this paper we focus on concurrent systems developed using common object-oriented multithreaded programming languages, in particular, Java. Specifically, we provide an algorithm to *observe* execution traces of multithreaded programs and, based on appropriate code instrumentation that allows one to atomically extract a partial-order causality from a linear sequence of events, we predict other schedules that are compatible with the run. The technique uses a weak *happens-before* relation which orders a write of a shared variable with all its subsequent reads that occur before the next write to the variable. A permutation of the observed events is a *possible execution* of a program if and only if it does not contradict the weak happens-before relation. Even though an observed execution trace may not violate the given specification, our algorithm infers other possible executions (consistent with the observed execution) that violate the given specification, if such an execution exists. Therefore, it can *predict* concurrency errors from non-violating runs.

1 Introduction

In multithreaded systems, threads can execute concurrently communicating with each other through a set of shared variables, creating the potential for subtle errors. The large number of potential interleavings makes it infeasible to check all possible executions before deployment. Ordinary testing of such systems, on the other hand, can be quite ineffective in practice, because of its low coverage with respect to the number of interleavings and because of the difficulty to reproduce many concurrency errors. The work presented in this paper builds upon our experience with *predictive runtime analysis* (or *predictive testing*) techniques, whose aim is to increase the effectiveness of testing by analyzing a class of possible executions that are causally equivalent to the particular observed one. What makes predictive analysis techniques appealing is the fact that some of the causally equivalent executions may violate the requirements of the system even though the observed execution does not.

Unlike model checking, predictive monitoring is not comprehensive. However, it is far more efficient than model checking because it does not execute the program but relies only on the information that is already available in a run-time execution. Specifically, we use a relatively non-restrictive semantic precedence relation, extracted entirely automatically at runtime via appropriate program instrumentation, to cluster events into equivalence classes. We then allow permutations of these equivalence classes and show how these permutations can be used to determine the effect of a large number of alternate schedules of threads.

```
Initially permit=f and landing=f;
Thread t1:                          Thread t2:

permit=checkTower();                permit=f;
if(permit)
   landing=t
else
   landing=f
```

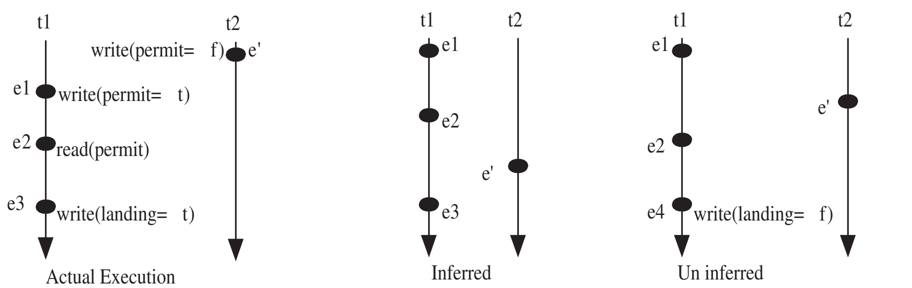

Fig. 1. Time increases downward and is assumed to be the same across the threads.

Example. Consider an execution of the multithreaded program in Figure 1 for airplane landing. Suppose in an execution, one thread (`t2`) in the program sets the variable `permit` to `false` (event e'). Another thread (`t1`) in the program checks with the control tower to see if the plane has permission to land. It then sets a variable `permit` to `true` (event e_1). At a subsequent point, the thread `t1` reads the variable `permit` (event e_2), checks if `permit` is `true`, and sets the variable `landing` to `true` (event e_3).

Suppose we want to check that the property that "if `landing` then immediately before `permit` is `true`". For the observed execution e', e_1, e_2, e_3, the property holds. However, since there is no causal connection between e' and e_1, and they are executed by different threads, we may permute these writes. Permuting only the writes would require us to actually execute the program along a different path (as in model checking). This would be inefficient and generally not feasible at runtime. We avoid doing so by requiring all associated reads (i.e., all reads of a variable that follow the latest preceding write of the variable) to also be permuted. This allows us to construct an alternate execution path, e_1, e_2, e', e_3 and the monitor infers that the property could be violated at e_3 and produces the trace as a witness.

Observe that, given the semantics of the program, the order of events could also have been: $e_1, e', e_2, ..$ in which case `landing` would never be set to `true`. We do not infer this path because doing so would require actually running the program with a different schedule (or semantically analyzing it) to determine which event happens instead of e_3. In particular, this means that violations of some properties may never be detected. For example, consider the property that "if `landing` is modified then `landing` is true or always in the past `permit` was false". This property is not violated by either the execution we observed, nor the alternate execution we constructed. However, it would be violated by the execution trace $e_1, e', e_2, ..$ and a model checker could detect this and our method could not, unless a related trace was one of the test cases. However, we show that our generalized analysis can very efficiently uncover many errors that standard testing would not with the same set of test cases.

2 Related Work

A number of runtime monitoring tools have been developed. These tools include NASA's JPaX [10], University of Pennsylvania's Java-MaC [13], Bell Labs' PET [9], and the commercial analysis systems Temporal Rover and DBRover [6, 7]. Our work builds on experience with related techniques and tools–namely, Java PathExplorer (JPaX) [10] and its sub-system Eagle [2]. These tools treat the execution of a program essentially as a flat, sequential trace of events or states. We proposed *predictive runtime analysis* in [17, 18]. The technique was based on checking a specification against executions that are *causally consistent* with a given execution – i.e., executions that do not permute writes to the same shared variable.

In this paper, we have significantly extended the strength of our prediction by abstracting a multithreaded computation in terms of two novel relations: *weak-happens-before* relation and *atomicity* relation on post-write set of read events. As a consequence of abstracting multithreaded computations this way, we are able increase the coverage of runtime analysis of multithreaded programs by being able to predict more valid multithreaded runs from a given single execution. In particular, in the example described above, the existing predictive technique would not have detected a violation of our specification.

3 Monitors for Safety Properties

Safety properties form an important class of properties 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. Nondeterministic automata are a standard means to succinctly represent languages over finite words. We next define a suitable version of automata, called *monitor*, with the property that it has a "bad" state from which it never exits:

Definition 1. *Let E be a finite or infinite set, that can be thought of as the set of events generated by the program to monitor. Then an E-*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*
- *$\rho\colon \mathcal{M} \times E \to 2^{\mathcal{M}}$ is a non-deterministic transition relation with the property that $\rho(b, e) = \{b\}$ for any $e \in E$.*

Sequences in $E^\star$, 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 $\rho\colon 2^{\mathcal{M}} \times E^\star \to 2^{\mathcal{M}}$ is recursively defined as $\rho(M, \epsilon) = M$ and $\rho(M, \pi e) = \rho(\rho(M, \pi), e)$, where $\rho\colon 2^{\mathcal{M}} \times E \to 2^{\mathcal{M}}$ is defined as $\rho(\{m\} \cup M, e) = \rho(m, e) \cup \rho(M, e)$ and $\rho(\emptyset, e) = \emptyset$, for all finite $M \subseteq \mathcal{M}$ and $e \in E$.*

$\mathcal{M}$ 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 [16, 11, 2], thus requiring the user to focus on system's formal safety requirements rather than on low level implementation details.

Example 1. Let us consider the program given in Figure 2. It consists of two threads `t1` and `t2` accessing the variables `x, y,` and `z`. Let the safety property that we want to monitor be "if `x` becomes positive then eventually in the past `x` became negative" which can be written in past-time temporal logic as the formula $F = p \to \Diamond q$, where p represents the event that `x` becomes positive and q represents the event that `x` becomes negative. The monitor automaton for this formula is given in Figure 2. State 4 in this automaton represents the bad state. Suppose that one runs the program and observes the execution `t1: x=-1; t1: z=x+3; t2: x=1; t2: y=x+z;` in that order; then, the safety property is not violated for this execution. Moreover, with the "happens-before" relation given in [17, 18] which disallows any permutation of two accesses of the same variable except when both of them are reads, one cannot predict any other possible valid run (obtained through a different scheduling) that violates the property. However, as shown later in this paper, our approach allows an observer of the execution above to predict another possible valid run that violates the safety property, namely the one in which `t2` executes first. The interesting aspect here is that the observer *does not see the code*, but only the flat sequence of read and write events of shared variables, time-stamped appropriately.

```
Initially x=0, y=0, z=0;
Thread t1:        Thread t2:

 x = -1;           x = 1;
 z = x+3;          y = x+z;
```

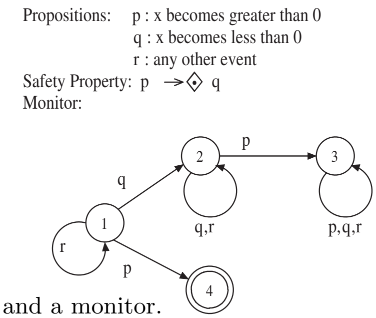

Fig. 2. Two threads `t1` and `t2` and a monitor.

4 Abstracting Multithreaded Computations

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. The computation of each thread is abstracted out in terms of *events*, while the multithreaded computation is abstracted out in terms of a partial order $\prec$ on events. There can be three types of events: an *internal* event, a *read* or a *write* of a shared variable. Internal events can be reads or writes of local variables, calling a function, the value of a variable crossing some threshold, etc. 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 and let S be the set of all variables that are shared by more than one thread in the execution.

We can define a special "happens-before" relation over the accesses to each shared variable: we say e x-happens-before e', written $e \lessdot_x e'$, iff e is a write of x and e' is a read of x such that the latest write to x that happens-before e' is e. In other words, we say that $e \lessdot_x e'$ if and only if the value of x read by event e' is the value written by the event e on variable x. This can be realized by maintaining a counter for each shared variable, which is incremented at each variable write. If the value of the counter at the read event e' of x is same as the counter value after the write event e of x, we say that $e \lessdot_x e'$. Let E_i denote the set of events of thread t_i and let E denote $\bigcup_i E_i$. Also, let $\lessdot \subseteq E \times E$ be defined as follows:

1. $e \lessdot e'$ when e and e' are events of *the same thread* and e happens immediately before e';
2. $e \lessdot e'$ whenever there is an $x \in S$ with $e \lessdot_x e'$.

The partial order $\prec$ is the transitive closure of the relation $\lessdot$. Let $\preccurlyeq$ be the transitive, reflexive closure of $\lessdot$. We say $e||e'$ if $e \not\preccurlyeq e'$ and $e' \not\preccurlyeq e$, i.e., the events e and e' are causally unrelated. The partial order $\prec$ captures a special causal "happens-before" relation among the events in different threads, which we call *weak-happens-before*. This causality relation is called "weak" since it is less constrained than the apparently more natural "happens-before" relation defined and investigated in [17, 18], which assumed that $e \lessdot_x e'$ also when e was a read of x and e' was a write of x or when both e and e' were writes of x; we call the causality in [18] apparently more natural since it captures exactly the common intuition that any two unrelated read accesses to a variable can be permuted.

While the causality in [18] allowed JMPAX to have strong predictive capabilities, the weak-happens-before causality considered in this paper significantly increases the coverage of runtime analysis of multithreaded systems and implicitly the predictive strength of tools implementing it, by allowing more possible runs to be inferred from just one observed execution of the system. All these predicted runs can occur under different thread scheduling or interleavings, meaning that *the increase in coverage comes at no expense*, that is, our technique is still free of false alarms. The fact that there are more possible execution traces to analyze must be clearly regarded as an advantage in the context of predictive runtime analysis; if, in the context of a highly unsynchronized multithreaded program, one finds the number of possible runs too large to analyze effectively, then one has the option to discard online as many of those "uninteresting" runs as needed. JMPAX already provides this functionality by allowing its users to tune an analysis breadth "knob", ranging from only one possible execution (the observed one), like in testing, to all possible executions, like in model-checking.

Unlike in [18], the weak-happens-before relation above is *not sufficient* to completely describe the multithreaded computation; if e and e' are two events such that $e \lessdot_x e'$ and e'' is another event writing x such that $e''||e$ and $e''||e'$, one *cannot* interleave e'' between e and e'. This is because if e'' happens in between e and e', then by the definition of $\lessdot_x$, it is the case that $e'' \lessdot_x e'$, which contradicts $e''||e'$. This observation suggests that given a write event, say e, of x, the set $\{e\} \cup \{e' \mid e' \in E \wedge e \lessdot_x e'\}$ should be regarded as *atomic* with respect to any other event outside the set that reads or writes x. Such a set is called an *atomic set* for the variable x. Therefore, each atomic set of $x \in S$ contains exactly one write and the corresponding reads. Any set which is a proper subset of an atomic set is called an *incomplete* atomic set. The atomic sets define another relation, called *atomicity relation* over the set of events E. We say that two events e and e' are *x-atomically related*, denoted by $e \Updownarrow_x e'$, if and only if e and e' belong to the same "atomic set" for the variable x. Formally, $e \Updownarrow_x e'$ if an only if there exists an event e'' such that both e and e' belong to the set $\{e''\} \cup \{e''' \mid e'' \lessdot_x e'''\}$. Therefore, $\Updownarrow_x$ is an equivalence relation on E. Let $[e]_x$ denote the corresponding *atomic equivalence class* of an event $e \in E$.

The structure described by $\mathcal{C} = (E, \prec, \Updownarrow)$ is called a *multithreaded computation*. A possible linearization of the events in E is *consistent* with $\prec$ if for any two events e and e' in E, $e \prec e'$ implies that e appears before e' in the linearization. Similarly, a linearization of the events in E is consistent with $\Updownarrow$ if for any two events e and e' and an arbitrary shared variable x, $e \Updownarrow_x e'$ implies that any other access (read or write) event e'' of x, such that $e'' \not\Updownarrow_x e$, appears either before or after both e and e' in the linearization. Combining the two conditions, we say that a linearization of the events E is consistent with a multithreaded computation $\mathcal{C} = (E, \prec, \Updownarrow)$ if and only if it is consistent with both $\prec$ and $\Updownarrow$.

Any such linearization of events consistent with the multithreaded computation 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 semantics of the multithreaded program. However, multiple consecutive writes of the same variable can be permuted provided that the set of a write and all reads following the write occur atomically. As seen in Section 6, by allowing an observer to analyze *multithreaded computations* rather than just *multithreaded executions*, one gets the benefit of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling.

5 Capturing Multithreaded Computations

To capture and transmit to an external observer the weak-happens-before and atomicity relations in a multithreaded computation, we use data-structures such as *vector clocks* and *atomicity identifiers*, respectively, as explained below. The algorithm based on vector clocks, which correctly and efficiently implements the weak-happens-before relation, is motivated by related work [8, 3, 14, 1]. However, the vector clock algorithm described in this paper differs from the algorithms described in previous works, because our focus here is to implement a different, less usual but more powerful w.r.t. monitoring "happens-before" relation. Let a vector clock $V : \mathit{ThreadId} \to \mathit{Nat}$ 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 presentation, we assume that each DVC V is a total map, where $V[t] = 0$ when 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 a DVC V_x with every shared variable x. 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.

Further, we associate a counter, called *atomicity identifier*, with every shared variable. Let c_x denote the counter associated with a shared variable x. These counters are initialized to 0. An atomicity counter associated with a variable keeps track of its atomic sets. A set of events corresponding to a read or write of x belong to an atomic set if and only if the atomicity identifiers associated with the variable x at those events are the same.

At every event in the multithreaded computation the DVCs and the atomicity identifiers are updated according to the following algorithm, which acts as a program instrumentation technique to emit events to an external observer of the system. If a thread t_i with current DVC V_i processes event e_i^k then

1. $V_i[i] \leftarrow V_i[i] + 1$;

2. if e_i^k is a write of a shared variable x then
   $V_x \leftarrow V_i$
   $c_x \leftarrow c_x + 1$;
3. if e_i^k is a read of a shared variable x then
   $V_i \leftarrow \max\{V_i, V_x\}$;
4. if e_i^k is a read or write of a shared variable x
   then send message $\langle e_i^k, i, V_i, x, c_x \rangle$ to observer
   else send message $\langle e_i^k, i, V_i, \perp, -1 \rangle$ to observer.

Intuitively, at every write event of a shared variable x, the DVC of x is updated with the DVC of the thread writing x. Thus, the thread passes its current time-stamp to the variable. This ensures that every event of the thread t_i till e_i^k happens before any event that reads the value written to x. The atomicity identifier is incremented by 1 to indicate that a new atomic set is starting; all the following read events, before another write of the same variable, will share the same atomicity identifier. At a read event of a variable x, the DVC of the reading thread is updated with the maximum of the DVC of the thread and the DVC of the variable x. This ensures that the read event happens after any previous event of the thread and the last write event of the variable x.

Theorem 1. *After event e_i^k is processed by thread t_i,*
a) $V_i[j]$ equals the number of events of t_j that "weak-happens-before" e_i^k; if $j = i$ then this number is k;
b) $V_x[j]$ is the number of events of t_j that "weak-happens-before" the most recent write of x; if $i = j$ and e_i^k is a write of x then this number also includes e_i^k.
Therefore, if $\langle e, i, V, x, c \rangle$ and $\langle e', j, V', x', c' \rangle$ are different messages sent by the algorithm, then $e \prec e'$ if and only if $V[i] \leq V'[i]$; if i and j are not given, then $e \prec e'$ if and only if $V < V'$. Moreover, $e \Updownarrow_x e'$ if and only if $x = x' \neq \perp$ and $c_x = c'_{x'}$.

Therefore, the code instrumentation algorithm above correctly implements the weak-happens-before and the atomicity relations.

6 Runtime Model Generation and Predictive Analysis

We now consider what happens at the observer's site, which receives messages $\langle e, i, V, x, c \rangle$ from the running multithreaded program, and which, because of Theorem 1, can infer the weak-happens-before and atomicity relations on these events. The observer can effectively, *online* and *in parallel.* analyze all possible interleavings of events that are consistent with the weak-happens-before and atomicity relations. Only one of these corresponds to the real execution. Since the other interleavings correspond to other possible executions, the presented technique has the capability to *predict* violations from successful executions.

6.1 Multithreaded Computation Lattice

Inspired by [1], we show how to incrementally generate an abstract model from a multithreaded computation, the *computation lattice*, with the properties: (1) every path in the computation lattice corresponds to a consistent multithreaded run; (2) every node in the computation lattice represents a set of events that can be observed as a prefix of a consistent multithreaded run. Our purpose in this paper is to check safety requirements against *all consistent* multithreaded runs of a system by systematically and efficiently exploring the computation lattice.

Let us fix an arbitrary multithreaded computation $\mathcal{C} = (E, \prec, \not\Updownarrow)$. Let e_i^k be the k^{th} event generated by the thread t_i since the start of its execution. A *cut* Σ is a subset of E such that for all $i \in [1, n]$, if $e_i^k \in \Sigma$ then $e_i^l \in \Sigma$ for all $l < k$. Let $\Sigma^{k_1 k_2 \ldots k_n}$ denote the cut containing the latest events $e_1^{k_1}, e_2^{k_2}, \ldots, e_n^{k_n}$ from each of the threads. If a thread i has not seen any event then k_i is considered 0.

Definition 2 (Consistent Cut). *A cut Σ is* ***consistent*** *if for all $e, e' \in E$,*
(a) if $e \in \Sigma$ and $e' \prec e$ then $e' \in \Sigma$, and
(b) if $e, e' \in \Sigma$ and $e \not\Updownarrow_x e'$ for some $x \in S$, then $[e]_x \subseteq \Sigma$ or $[e']_x \subseteq \Sigma$.

(a) says that a consistent cut is closed under the weak-happens-before relation, and *(b)* says that a consistent cut can contain at most one incomplete atomic set for any shared variable. Indeed, if *(b)* fails, then there is no way to reorder the remaining events in $E - \Sigma$ without violating the atomicity relation.

Definition 3. *An event e_i^l is* ***enabled*** *for a consistent cut $\Sigma = \Sigma^{k_1 k_2 \ldots k_n}$ iff*
(a) $l = k_i + 1$,
(b) for all events $e \in E$, if $e \prec e_i^l$ then $e \in \Sigma$, and
(c) if e_i^l is an access (read or write) event of an $x \in S$ and e is any access event of x in Σ then either $e_i^l \in [e]_x$ or $[e]_x \subseteq \Sigma$.

Since e_i^l can be in at most one atomic set for a given shared variable, the above actually says that e_i^l can be safely considered a next event in the execution. Indeed, the following can be regarded as an equivalent definition of enabledness:

Proposition 1. *e_i^l is enabled for a consistent Σ iff $\Sigma \cup \{e_i^l\}$ is also consistent.*

Proof. Since Σ is a cut, all the events $e_i^1, e_i^2, \ldots, e_i^{k_i}$ are in Σ. Therefore, $\Sigma \cup \{e_i^l\}$ contains all events e_i^m, for $m < l$, if $l = k_i + 1$. This implies that $\Sigma \cup \{e_i^l\}$ is a cut. Since Σ is a consistent cut, for all events $e \in \Sigma$, if $e' \prec e$ then $e' \in \Sigma$. It is given that for all events $e' \prec e_i^l$, $e' \in \Sigma$. Therefore, for all events $e \in \Sigma \cup \{e_i^l\}$, if $e' \prec e$ then $e' \in \Sigma$. This is the first condition for $\Sigma \cup \{e_i^l\}$ being a consistent cut. Let e be any access event of x in Σ. Given that Σ is a consistent cut, if $e_i^l \in [e]_x$ then the second condition for the definition of consistent cut continues to hold for $\Sigma \cup \{e_i^l\}$ because the addition of e_i^l to Σ cannot create a new atomic set for x. Otherwise, if $e_i^l \notin [e]_x$ then we know that $[e]_x \subseteq \Sigma$. This implies that $[e]_x \subseteq \Sigma \cup \{e_i^l\}$ or $[e_i^l]_x \subseteq \Sigma \cup \{e_i^l\}$. Hence, the second condition for the definition of consistent cut holds for $\Sigma \cup \{e_i^l\}$. Since both the first and second conditions for the definition of consistent cut holds for the cut $\Sigma \cup \{e_i^l\}$, $\Sigma \cup \{e_i^l\}$ is a consistent cut. $\square$

Definition 4. *If $\Sigma = \Sigma^{k_1 k_2 \ldots k_n}$ is consistent and e_i^l is enabled for Σ, then let $\delta(\Sigma, e_i^l)$ denote the consistent cut $\Sigma \cup \{e_i^l\}$, that is, $\Sigma^{k_1 k_2 \ldots k_{i-1} (k_i+1) k_{i+1} \ldots k_n}$.*

Therefore, δ maps a consistent cut Σ and a corresponding enabled event e into another consistent cut, which can be regarded as the result of executing e after executing all the events in Σ in some consistent way. Let $\Sigma^{K_0} = \Sigma^{00\ldots0}$ be the consistent cut at the beginning of the computation. Then

Proposition 2. *A consistent multithreaded run $R = e_1e_2\ldots e_{|E|}$ generates a sequence of consistent cuts $\Sigma^{K_0}\Sigma^{K_1}\ldots\Sigma^{K_{|E|}}$ such that for all $r \in \overline{1,|E|}$, $\Sigma^{K_{r-1}}$ is a consistent cut, e_r is enabled for $\Sigma^{K_{r-1}}$, and $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$.*

Proof. The proof is by induction on r. By definition Σ^{K_0} is a consistent cut. Moreover, it is easy to see that e_1 is enabled in Σ^{K_0}. Since Σ^{K_0} is a consistent cut 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 a consistent cut, e_r is enabled in $\Sigma^{K_{r-1}}$, and $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$. Therefore, by Proposition 1, $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$ is also a consistent cut. Let $\Sigma^{K_r} = \Sigma^{k_1k_2\ldots k_n}$ and $C = \Sigma^{K_r}$. We want to prove that e_{r+1} is enabled in Σ^{K_r}. Let $e_{r+1} = e_i^l$ for some i and l i.e. e_{r+1} is the l^{th} event of thread t_i. For every event e_i^k, such that $k < l$, $e_i^k \prec e_i^l$. Therefore, by the definition of consistent run, in R, 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. Therefore, $k_i = l - 1$. Thus the first condition for e_{r+1} being enabled for Σ^{K_r} is met. Since C is a consistent cut, for all events e and e', if $e \neq e_i^l$ then $(e \in C \cup \{e_i^l\}) \wedge (e' \prec e) \rightarrow e' \in C \cup \{e_i^l\}$. Otherwise, if $e = e_i^l$ then by the definition of consistent run, if $e' \prec e_i^l$ then e' appears before e_i^l in R. This implies that e' is included in $C \cup \{e_i^l\}$. Therefore, for all events e and e', if $e \in C \cup \{e_i^l\}$ and $e' \prec e$ then $e' \in C \cup \{e_i^l\}$. Thus the second condition for e_{r+1} being enabled for Σ^{K_r} is met. Let e_{r+1} be access event of a shared variable x. Let e be an event in the incomplete atomic set (if exists) for x in C. If $e \Updownarrow_x e_{r+1}$, the third condition for the enabledness of an event is not violated. If $e \not\Updownarrow_x e_{r+1}$ and $\exists e' \in E - (C \cup \{e_i^l\})$ such that $e \Updownarrow_x e'$ then any run that extends $e_1e_2\ldots e_{r+1}$ will be inconsistent with respect to the "atomicity" relation. Therefore, if $e \not\Updownarrow_x e_{r+1}$ then $[e]_x \subseteq C \cup \{e_{r+1}\}$. Thus the third condition for e_{r+1} being enabled for Σ^{K_r} is met. Therefore, we proved that e_{r+1} is enabled for the consistent cut Σ^{K_r}. Since, Σ^{K_r} is a consistent cut and e_{r+1} is enabled in Σ^{K_r}, $\delta(\Sigma^{K_r}, e_{r+1})$ is defined. We let $\delta(\Sigma^{K_r}, e_{r+1}) = \Sigma^{K_{r+1}}$. $\square$

From now on, we identify sequences $\Sigma^{K_0}\Sigma^{K_1}\ldots\Sigma^{K_{|E|}}$ as above with multithreaded runs, and simply call them *runs*. We say that Σ *leads-to* Σ', written $\Sigma \rightsquigarrow \Sigma'$, when there is some run in which Σ and Σ' are consecutive consistent cuts. Let $\rightsquigarrow^*$ be the reflexive transitive closure of the relation $\rightsquigarrow$. The set of all consistent cuts together with the relation $\rightsquigarrow^*$ forms a *lattice* with n mutually orthogonal axes representing each thread. For a consistent cut $\Sigma^{k_1k_2\ldots k_n}$, we call $k_1 + k_1 + \cdots k_n$ its *level*. A *path* in the lattice is a sequence of consistent cuts where the level increases by 1 between any two consecutive consistent cuts 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 events of thread t_i in the multithreaded computation. This lattice, called *computation lattice*, can be regarded as an *abstract model* of the running multithreaded program.

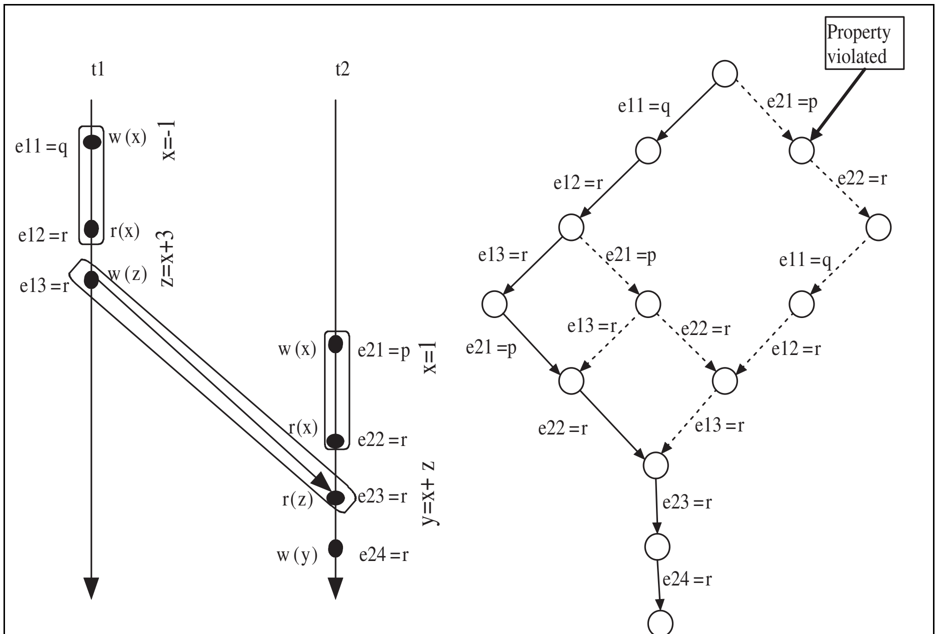

Fig. 3. Successful Execution and Computation Lattice.

Figure 3 shows the weak-happens-before and atomicity relations on the events generated by the multithreaded execution in Example 1, together with the corresponding computation lattice. The rectangular boxes enclose the atomic sets $\{e_1^1, e_1^2\}$, $\{e_1^3, e_2^3\}$, and $\{e_2^1, e_2^2\}$. The actual execution is marked with solid edges in the lattice. It can be readily seen that the temporal property defined in Example 1 holds on the actual execution of the program, but that it is violated on some other consistent run represented by the sequence of events $e_2^1, e_2^2, e_1^1, e_1^2, e_1^3, e_2^3, e_2^4$.

6.2 Level By Level Analysis of the Computation Lattice

A naive observer of a multithreaded program would just check the observed execution trace against the monitor for the safety property, say $\mathcal{M}on$, and would maintain at each moment a set of states, say *MonStates*, in $\mathcal{M}on$. When a new event e arrives, it would replace *MonStates* by $\rho(\mathit{MonStates}, e)$. If the bad state b occurs 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.

A smart observer, as seen next, analyzes 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 program 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 an algorithm that a smart observer can use to construct and traverse the computation lattice.

The observer maintains a set of consistent cuts, (*CurrLevel*), that are present in the current level of the lattice. For each event e in Q, it tries to construct a new consistent cut from the set of consistent cuts in the current level and the event e. If the consistent cut is created successfully then it is added to the set of consistent cuts (*NextLevel*) for the next level of the lattice. The process continues until no more consistent cut can be created for the next level. At that time, the current level is complete and the observer starts constructing the next level by setting *CurrLevel* to *NextLevel* and *NextLevel* to the empty set, and reallocating the space occupied by *CurrLevel*. Fig. 4 shows the pseudo-code.

```
while(not empty(Q)){
  monitorLevel()
}

State cut(Σ, m, Q){
  create Σ' such that
    VC(Σ') = VC(Σ)
    and AI(Σ') = AI(Σ)
  let m is of the form ⟨e, i, V, x, c⟩
  VC(Σ')[i] ← VC(Σ)[i] + 1
  if c ≥ 0 and
    ∃⟨e', i', V', x, c'⟩ ∈ Q such that
    V' ≰ max(VC(Σ), V) and c = c'{
      AI(Σ')[x] ← c
    }
  for each s ∈ M(Σ){
    M(Σ') ← M(Σ') ∪ ρ(s, e)
    if b ∈ M(Σ')
      output 'property violated'}
  return Σ'
}

boolean monitorLevel(){
  for each m ∈ Q and Σ ∈ CurrLevel{
    if enabled(Σ, m) {
      NextLevel ← NextLevel ⊎ cut(Σ, m, Q)}}
  Q ← removeUselessMessages(CurrLevel, Q)
  CurrLevel ← NextLevel
  NextLevel ← ∅
}

boolean enabled(Σ, m){
  let m is of the form ⟨e, i, V, x, c⟩
  if not(∀j ≠ i : VC(Σ)[j] ≥ V[j] and
        VC(Σ)[i] + 1 = V[i]) return false
  if c ≥ 0 and AI(Σ)[x] ≥ 0 and AI(Σ)[x] ≠ c{
      return false
  }
  return true
}
```

Fig. 4. Level-by-level traversal.

Every consistent cut Σ contains a set of monitor states $\mathcal{M}(\Sigma)$, a DVC $VC(\Sigma)$ to represent the latest events from each thread that resulted in that consistent cut, and an atomic identifier map $AI(\Sigma)$ that maps every shared variable x to the atomic identifier corresponding to the incomplete atomic set in Σ for $x \in S$, if it exists, or to -1 if there is no incomplete atomic set for x in Σ. The predicate $enabled(\Sigma, m)$, checks if the event e contained in the message m is enabled in the consistent cut Σ. For that, it first checks if for every event $e' \in \Sigma$, $e' \prec e$, by comparing the DVCs. If this is not the case then $enabled(\Sigma, m)$ returns false. Otherwise, it checks if the atomic identifier of e matches the atomic identifier of the incomplete atomic set, if it exists, for the shared variable x. If this is not the case, then $enabled(\Sigma, m)$ returns false; otherwise returns true. The correctness of the function follows from Theorem 1 and the definition of enabledness of an event for a consistent cut. It essentially says that event e can generate a consecutive consistent cut from the consistent cut Σ iff 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 $cut(\Sigma, m, Q)$, which implements the function δ in Definition 4, creates a new consistent cut Σ', as the consistent cut resulting from Σ after adding the event e of message m. It first copies the DVC and the atomic identifier map associated with Σ to Σ'. Then it increments the i^{th} element of the DVC of Σ' and updates the atomic identifier map of Σ' for variable x with the atomic identifier of e if Σ' still contains an incomplete atomic set for x. For every monitor state s in $\mathcal{M}(\Sigma)$, it applies the monitoring function ρ to s and e and adds the resulting states in the set $\mathcal{M}(\Sigma')$. After the update, if $\mathcal{M}(\Sigma')$ contains the bad state b then a `'property violated'` error is raised.

The merging operation $nextLevel \uplus \Sigma$ adds the consistent cut Σ to the set *nextLevel*. If Σ is already present in *nextLevel*, it updates the existing cut's *MonStates* with the union of the existing state's *MonStates* and the *Monstates* of Σ. Two consistent cuts are the same if their DVCs are equal. The function *removeUselessMessages(CurrLevel,Q)* removes from Q all the messages that cannot contribute to the construction of any cut at the next level. To do so, it creates a DVC V_{min} for which each component is the minimum of the corresponding component of the DVCs of all the consistent cuts in the set *CurrLevel*. It then removes all the messages in Q whose DVCs are less than or equal to V_{min}. This function makes sure that we do not store any unnecessary messages.

6.3 Handling Synchronization Constructs

In Java, one can synchronize blocks of statements by using the keyword `synchronize` with an object over which the block is synchronized. When the execution enters the synchronized block, it acquires the lock associated with the object and releases the lock when it exits the block. The main goal of synchronization is to attain atomicity: if two synchronized blocks over the same lock are executed by two different threads, then their execution cannot be interleaved. This atomicity can be naturally achieved in our approach by generating dummy write and read events of the lock variable when the lock is acquired or released, respectively. In particular, since in Java synchronized blocks holding the same lock cannot be interleaved, so corresponding events cannot be permuted, locks are considered shared variables and a write event of a lock is generated whenever a lock is acquired, and a read event of the lock is generated whenever a lock is released. This way, we make a block holding a lock atomic with respect to any other block holding the same lock, thus avoiding reporting any false alarms.

7 Application to Data-Race Detection

Since the predictive runtime analysis approach discussed in the previous sections is parameterized by a very generic concept of monitor as a nondeterministic finite state machine of bad prefixes, it can be applied to predict violations of requirements specifications given in a variety of formalisms. Temporal logics and regular expressions are just special cases. In particular, our technique can be used as a complementary approach to model-checking, when the total number of states to be model-checked is prohibitively large.

We next discuss another interesting application of our runtime analysis technique, namely in predicting data-races from data-race-free executions. The idea is to specify some simple temporal logic formulae, which, if violated, imply the existence of data-races in a multithreaded computation. A data-race occurs when two threads access a shared variable simultaneously without any synchronization and at least one of the accesses is a write. Data-races can lead to very unexpected behaviors of concurrent systems, and are notorious for their difficulty to detect. Plain testing can easily escape data-races, due to their dependency on thread-scheduling. For example, suppose that two threads increment a shared variable x simultaneously by executing statements $x{+}{+}$ without any synchronization. If the initial value of x is 0 then at the end of the execution the value of x can be 1 or 2. The former is obviously wrong, but hard to catch during testing.

It has been broadly recognized that tools capable of detecting data-races automatically in programs at runtime can be very valuable. There has been a substantial effort dedicated to developing tools and techniques that detect data-races online, such as those based on "happens-before" relations over locks [5], or those based on *locksets*, such as Eraser [15]. We next show how one can use our predictive runtime analysis technique to *precisely* detect data-races in a way somewhat similar to [5]. An advantage of our technique over the former approaches based on "happens-before" causality, such as the one in [5], is that we can *permute two synchronized blocks* holding the same lock due to our less constrained weak-happens-before relation. For example, if one sees the execution trace `t1: z=1; t1: lock(l); t1: x=0; t1: unlock(l); t2: lock(l); t2: y=10; t2: unlock(l); t2: z=0;`, then the "happens-before" data-race detection algorithm in [5] cannot detect the potential data-race over the variable `z`. However, it is easy to see that our approach can construct the consistent run `t2: lock(l); t2: y=10; t2: unlock(l); t2: z=0; t1: z=1; t1: lock(l); t1: x=0; t1: unlock(l);` that exhibits the data-race over `z`. Moreover, since we do the analysis at runtime, we can take a necessary recovery action whenever we find a data-race.

We conservatively say that two accesses of a shared variable x, of which at least one is a write, by two threads are *in data-race conflict*, if one can permute events consistently with the multithreaded computation such that the two accesses become consecutive events. Using our predictive monitoring approach, one can detect such data-race conflicts by monitoring the following simple property for every shared variable x and for every pair of threads t_i and t_j:

$$\begin{aligned}(\mathrm{write}(x,t_i) \rightarrow \neg \odot \mathrm{write}(x,t_j)) \;&\wedge (\mathrm{write}(x,t_i) \rightarrow \neg \odot \mathrm{read}(x,t_j)) \\ &\wedge (\mathrm{read}(x,t_i) \rightarrow \neg \odot \mathrm{write}(x,t_j))\end{aligned}$$

where the temporal operator $\odot F$ means "F holds at the previous event", the events $\mathrm{read}(x,t)$ (or $\mathrm{write}(x,t)$) are generated whenever the thread t reads (or writes) x. The first conjunct in the formula states the absence of write-write data-races. A write-write data-race happens if there is a consistent run in which two different threads write a variable x consecutively. Similarly, the second and the third conjunct state the absence of write-read and read-write data-races. Using our approach, by monitoring the above formulae, one can detect data-races in multithreaded programs precisely, that is, without false positives.

8 Implementation

We have implemented this novel predictive runtime analysis technique as part of version 3.0 of the tool Java MultiPathExplorer (JMPaX) [12], designed to monitor multithreaded Java programs. The current implementation is written in Java and it removes the previous limitation of version 2.0 that all the shared variables are static and of type `int`. The tool has three main modules, the *instrumentation* module, the *observer* module and the *monitor* module.

The instrumentation module takes a specification file and a list of class files as command line arguments, and it instruments each class file provided as argument to send messages to the observer module whenever a relevant read, write, or internal event occurs at runtime. The instrumentation module uses the BCEL Java library [4] to modify Java class files.

The *observer* module generates the lattice level-by-level as the events are received from the instrumented program. The *monitor* module reads the requirements specification file, currently using either linear temporal logic or regular expression formalism, and generates the non-deterministic monitor corresponding to the bad prefixes of the specification. An implementation of the monitor transition function ρ is provided as an interface method to the *observer* module. This method raises an exception if at any point the set of states returned by ρ contains the "bad" state of the monitor. The system being modular, the user can plug in his/her own *monitor* module for his/her logic of choice.

9 Conclusion

We have developed a simple and efficient technique to predict violations of safety properties of concurrent object-oriented programs. Our algorithm requires maintaining an atomic identifier map for every consistent cut. The size of this map is linearly proportional to the number of shared variables. This can lead to consumption of a large amount of memory space if the number of shared variables is large and slow down the monitoring process. As an aside, this reinforces the view that avoiding unnecessary sharing of variables is good software practice; in this case, fewer variables will improve the efficiency of monitoring (as well as reduce the chances of errors). While our technique will not find all errors, it can be applied to detect important software errors such as unintended data-race conditions which may otherwise be missed. The technique is, however, sound: it does not produce any false positives (any errors predicted could actually occur in a different execution).

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. 1993.

2. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proc. of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57, 2004.
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. M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universitat at Berlin, Institut für Informatik, April 2001.
5. A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.
6. D. Drusinsky. Temporal Rover. http://www.time-rover.com.
7. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330, 2000.
8. 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.
9. E. L. Gunter, R. P. Kurshan, and D. Peled. PET: An interactive software testing tool. In *Proc. of Computer Aided Verification (CAV'00)*, volume 1885 of *LNCS*, pages 552–556, 2000.
10. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proc. of Workshop on Runtime Verification (RV'01)*, volume 55 of *ENTCS*, 2001.
11. 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 *LNCS*, pages 342–356, 2002.
12. Java MultiPathExplorer (JMPaX). Download: `http://fsl.cs.uiuc.edu/jmpax/`.
13. 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 *ENTCS*, 2001.
14. 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.
15. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
16. K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. In *Proc. of the 3rd Workshop on* Runtime Verification (RV'03), volume 89 of *ENTCS*, pages 162–181, 2003.
17. 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.
18. K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *Proc. of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 123–138, 2004.