Instrumentation Technique for Online Analysis of Multithreaded Programs

Grigore Roşu and Koushik Sen
Email: {grosu,ksen}@cs.uiuc.edu

Department of Computer Science, University of Illinois at Urbana-Champaign, Siebel Center, 201 N. Goodwin, Urbana, Illinois 61801, USA

SUMMARY

This paper presents an automatic code instrumentation technique, based on *multithreaded vector clocks*, for extracting the causal partial order on relevant state update events from a running multithreaded program. This technique is used in a formal testing environment, not only to detect, but especially to *predict safety errors* in multithreaded programs. The prediction process consists of rigorously analyzing other potential executions that are consistent with the causal partial order: some of these can be erroneous despite the fact that the particular observed execution was successful. The technique has been implemented as part of a Java program analysis tool.

KEY WORDS: runtime verification; multithreaded systems; vector clocks

1. Introduction and Motivation

A major drawback of testing is its lack of coverage: if an error is not exposed by a particular test case then that error is not detected. To ameliorate this problem, many techniques have been investigated to increase the coverage of testing, such as test-case generation methods for generating those test cases that can reveal potential errors with high probability [8, 20, 28]. Based on experience with related techniques already implemented in JAVA PATHEXPLORER (JPAX) [15] and its sub-system EAGLE [4], we have proposed in [25, 26] an alternative approach, called "predictive runtime analysis", which can be intuitively described as follows.

Suppose that a multithreaded program has a subtle safety error, such as a safety or a liveness temporal property violation, or a deadlock or a data-race. Like in testing, one executes the program on some carefully chosen input (test case) and suppose that, unfortunately, the error is not revealed during that particular execution; such an execution is called *successful* with respect to that bug. If one regards the execution of a program as a flat, sequential trace of events or states, like NASA's JPAX system [15], University of Pennsylvania's JAVA-MAC

[19], Bell Labs' PET [14], or the commercial analysis systems Temporal Rover and DBRover [9, 10, 11], then there is not much left to do to find the error except to run another, hopefully better, test case. However, by observing the execution trace in a smarter way, namely as a causal dependency partial order on state updates, one can predict errors that can potentially occur in other possible runs of the multithreaded program.

The present work is an advance in *runtime verification* [16], a scalable complementary approach to traditional formal verification (such as theorem proving and model checking [6]). Our focus here is on multithreaded systems with shared variables. We present a simple and effective algorithm that enables an external observer of an executing multithreaded program to detect and predict specification violations. The idea is to *instrument* the system before its execution, so that it will emit relevant events at runtime. No particular specification formalism is adopted in this paper, but examples are given using a temporal logic that we are currently considering in JAVA MULTIPATHEXPLORER (JMPAX) [25, 26], a tool for safety violation prediction in Java multithreaded programs which supports the presented technique.

In multithreaded programs, threads communicate via a set of shared variables. Some variable updates can causally depend on others. For example, if a thread writes a shared variable x and then another thread writes y due to a statement $y = x+1$, then the update of y *causally depends* upon the update of x. Only read-write, write-read and write-write causalities are considered, because multiple consecutive reads of the same variable can be permuted without changing the actual computation. A state is a map assigning values to variables, and a specification consists of properties on these states. Some variables may be of no importance at all for an external observer. For example, consider an observer which monitors the property "if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false". All the other variables except x, y and z are irrelevant for this observer (but they can clearly affect the causal partial ordering). To minimize the number of messages sent to the observer, we consider a subset of *relevant events* and the associated *relevant causality*.

We present an algorithm that, given an executing multithreaded program, generates appropriate messages to be sent to an external observer. The observer, in order to perform its more elaborated system analysis, extracts the state update information from such messages together with the relevant causality partial order among the updates. This partial order abstracts the behavior of the running program and is called *multithreaded computation*. By allowing an observer to analyze multithreaded computations rather than just flat sequences of events, one gets the benefit of not only properly dealing with potential reordering of delivered messages (reporting global state accesses), but also of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling and can be hard, if not impossible, to find by just testing.

To be more precise, let us consider a real-life example where a runtime analysis tool supporting the proposed technique, such as JMPAX, would be able to predict a violation of a property from a single, successful execution of the program. However, like in the case of data-races, the chance of detecting this safety violation by monitoring only the actual run is very low. The example consists of a two threaded program to control the landing of an airplane. It has three variables `landing`, `approved`, and `radio`; their values are 1 when the *plane is landing*, *landing has been approved*, and *radio signal is live*, respectively, and 0 otherwise. The

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

void thread2() {while(radio){checkRadio()}}
void checkRadio() {possibly change value of radio}
```

Figure 1. A buggy implementation of a flight controller.

safety property to verify is "If the plane has started landing, then it is the case that landing has been approved and since the approval the radio signal has never been down."

The code snippet for a naive implementation of this control program is shown in Fig. 1. It uses some dummy functions, `askLandingApproval` and `checkRadio`, which can be implemented properly in a real scenario. The program has a serious problem that cannot be detected easily from a single run. The problem is as follows. Suppose the plane has received approval for landing and just before it started landing the radio signal went off. In this situation, the plane must abort landing because the property was violated. But this situation will very rarely arise in an execution: namely, when `radio` is set to 0 between the approval of landing and the start of actual landing. So a tester or a simple observer will probably never expose this bug. However, note that even if the radio goes off *after* the landing has started, a case which is quite likely to be considered during testing but in which the property is *not* violated, JMPaX will still be able to construct a possible run (counterexample) in which radio goes off between landing and approval. In Section 4, among other examples, it is shown how JMPaX is able to predict two safety violations from a single successful execution of the program. The user will be given enough information to understand the error and to correct it. In fact, this error is an artifact of a bad programming style and cannot be easily fixed - one needs to give a proper event-based implementation. This example shows the power of the proposed runtime verification technique as compared to the existing ones in JPaX and Java-MaC.

The main contribution of this paper is a detailed presentation of an instrumentation algorithm which plays a crucial role in extracting the causal partial order from one flat execution, and which is based on an appropriate notion of vector clock inspired from [13, 22], called *multithreaded vector clock (MVC)*. An MVC V is an array associating a natural number $V[i]$ to each thread i in the multithreaded system, which represents the number of relevant events generated by that thread since the beginning of the execution. An MVC V_i is associated to each thread t_i, and two MVCs, V_x^a (access) and V_x^w (write) are associated to each shared variable x. When a thread t_i processes event e, which can be an internal event or a shared variable read/write, the algorithm $\mathcal{A}$ in Section 3 is executed. We prove that $\mathcal{A}$ correctly

implements the relevant causal partial order. This algorithm can be implemented in several ways. In the case of Java, we prefer to implement it as an appropriate instrumentation procedure of code or bytecode, to execute $\mathcal{A}$ whenever a shared variable is accessed. Another implementation could be to modify a JVM. Yet another one would be to enforce shared variable updates via library functions, which execute $\mathcal{A}$ as well.

ConTest [12] gives another technique to detect synchronization faults in multithreaded programs, based on controlling the Java scheduler at the source code level. The source of the program under test is seeded with a `sleep()`, `yield()`, or `priority()` primitive at shared memory accesses and synchronization events. At run time, ConTest makes random or coverage-based decisions as to whether the seeded primitive is to be executed. Unlike ConTest, which executes the program with different schedules, JMPaX predicts other possible runs from a single execution, *without re-executing the program*. Thus, ConTest has the potential to cover even execution traces that are not causally equivalent. However, except unavoidable program-observer communication delays, our technique is expected to lead to quite efficient runtime analysis tools, because it does not interfere with the normal execution of programs and it does not rely on re-executing programs: it needs only the information that is already available in a run-time execution to predict other interleavings.

Preliminary versions of the work in this paper has been presented at the workshop *Parallel and Distributed Systems: Testing and Debugging (PADTAD'04)* in April 2004 [23].

2. Multithreaded Systems

We consider multithreaded systems in which several threads communicate with each other via a set of shared variables. A crucial point is that some variable updates can causally depend on others. We will present an instrumentation algorithm which, given an executing instrumented multithreaded program, generates appropriate messages to be sent to an external observer. The observer, in order to perform its analysis, extracts the state update information from such messages together with the causality partial order among the updates.

In this paper we only consider a fixed number of threads. However, the presented instrumentation technique can be easily extended to systems consisting of a variable number of threads, where these can be dynamically created and/or destroyed [26].

Multithreaded executions. Given n threads t_1, t_2, ..., t_n, a *multithreaded execution* is a sequence of events $e_1 e_2 \ldots e_r$, each belonging to one of the n threads and having type *internal*, *read* or *write* of a shared variable. We use e_i^k to represent the k-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 may refer to it generically, such as e, e', etc. We write $e \in t_i$ when e is generated by t_i.

Let us fix an arbitrary but fixed multithreaded execution, say $\mathcal{M}$, and let S be the set of all shared variables. There is an immediate notion of *variable access precedence* for each shared variable $x \in S$: we say that e *x-precedes* e', written $e <_x e'$, if and only if e and e' are variable access events (reads or writes) to the same variable x, and e "happens before" e', that is, e occurs before e' in $\mathcal{M}$. This "happens-before" relation can be easily realized in practice by keeping a counter for each shared variable, which is incremented at each variable access.

The notion of precedence above is consistent with the *sequential memory model* of multithreaded systems, which is assumed from now on in this paper. In fact, we assume that all shared memory accesses (reads or writes of shared variables in S) are atomic and instantaneous. this will allow us to properly reason about causal dependencies between events. For example, if a thread writes x then writes y followed by another thread writing y then x, under a non-sequential memory model it would be possible that the first write of y takes place before the second, while the first write of x takes place *after* the second write of x, leading to a circular causal dependency between the two threads. So far we have not considered non-sequential memory models in our runtime verification efforts, which admittedly would reveal additional potential errors but would increase the complexity of runtime analysis. This may change in the future if the assumed sequential model will be found too restrictive in practical experiments. Recall that our purpose is to find errors in multithreaded programs, not to prove systems correct. The errors that we detect using the sequential model are also errors in other memory models, so our approach is currently conservative.

Causality and multithreaded computations. Let $\mathcal{E}$ be the set of all events occurring in the multithreaded execution $\mathcal{M}$ and let $\prec$ be the partial order on $\mathcal{E}$ defined as follows: $e_i^k \prec e_i^l$ if $k < l$; $e \prec e'$ if there is $x \in S$ with $e <_x e'$ and at least one of e, e' is a write; $e \prec e''$ if $e \prec e'$ and $e' \prec e''$. Thus, the events of a thread i are causally ordered by their corresponding occurrence time, and if two events e and e', of the same thread or not, access a shared variable x and one of them is a write, then the most recent one causally depends on the former one. No causal constraint is imposed on read-read events, so they are permutable. Finally, by closing it under transitivity, $\prec$ becomes the smallest partial order including the two causal constraints. We write $e||e'$ if $e \not\prec e'$ and $e' \not\prec e$. The partial order $\prec$ on $\mathcal{E}$ defined above is called the *multithreaded computation* associated with the original multithreaded execution $\mathcal{M}$.

As discussed later in more depth, synchronization of threads can be handled at no additional effort by just generating appropriate 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 linearization (or a permutation) of all the events e_1, e_2, ..., e_r that is consistent with 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*. Intuitively, a multithreaded run can be viewed as a possible execution of the same system under a different execution speed of each individual thread.

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 its semantics. Indeed, an external observer *cannot disregard* the order in which the same variable is modified and used within the observed execution, because this order can be part of the intrinsic semantics of the multithreaded program. However, multiple causally independent modifications of different variables can be permuted, and the particular order observed in the given execution is not critical. By allowing an observer to analyze *multithreaded computations*, rather than just *multithreaded executions* like JPaX [15], Java-MaC [19], and PET [14], one gets the benefit of not only properly dealing with potential re-orderings of delivered messages (e.g., due to using multiple channels to reduce the overhead), but also of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling.

Relevant causality. Some variables in S may be of no importance for an external observer. E.g., consider an observer whose purpose is to check "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 notation in [17], 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 [21] 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 $e \lhd e'$ if and only if $e, e' \in \mathcal{R}$ and $e \prec e'$. Note that the other shared variables can also indirectly influence the relation $\lhd$, because they can influence the relation $\prec$.

We next introduce multithreaded vector clocks (MVC), together with a technique that is proved to correctly implement the relevant causality relation.

3. Multithreaded Vector Clock Algorithm

Inspired and stimulated by the elegance and naturality of vector clocks [13, 22, 2] in implementing causal dependency in distributed systems, we next devise an algorithm to implement the relevant causal dependency relation in multithreaded systems. Since in multithreaded systems communication is realized by shared variables rather than message passing, to avoid any confusion we call the corresponding vector-clock data-structures *multithreaded vector clocks* and abbreviate them *(MVC)*. The algorithm presented next has been mathematically derived from its desired properties.

For each thread i, where $1 \leq i \leq n$, let us consider an n-dimensional vector of natural numbers V_i. Intuitively, the number $V_i[j]$ represents the event number at thread t_j that the thread t_i is "aware" of. Since communication in multithreaded systems is done via shared variables, and since reads and writes have different weights in our approach, we let V_x^a and V_x^w be two additional n-dimensional vectors for each shared variable x; we call the former *access MVC* and the latter *write MVC*. All MVCs are initialized to 0. As usual, for two n-dimensional vectors, $V \leq V'$ if and only if $V[j] \leq V'[j]$ for all $1 \leq j \leq n$, and $V < V'$ if and only if $V \leq V'$ and there is some $1 \leq j \leq n$ such that $V[j] < V'[j]$; also, $\max\{V, V'\}$ is the vector with $\max\{V, V'\}[j] = \max\{V[j], V'[j]\}$ for each $1 \leq j \leq n$. Our goal is to find a procedure that updates these MVCs and emits a minimal amount of events to an external observer, which can further efficiently extract the relevant causal dependency relation. Formally, the requirements of such a procedure, say $\mathcal{A}$, which works as an event filter, or an abstraction of the given multithreaded execution, must include the following natural requirements.

Requirements for $\mathcal{A}$. *After $\mathcal{A}$ updates the MVCs as a consequence of the fact that thread t_i generates event e_i^k during the multithreaded execution $\mathcal{M}$, the following should hold: (a) $V_i[j]$ equals the number of relevant events of t_j that causally precede e_i^k; if $j = i$ and e_i^k is relevant then this number also includes e_i^k; (b) $V_x^a[j]$ equals the number of relevant events of t_j that causally precede the most recent event in $\mathcal{M}$ that accessed (read or wrote) x; if $i = j$ and e_i^k is a relevant read or write of x event then this number also includes e_i^k; (c) $V_x^w[j]$ equals the number of relevant events of t_j that causally precede the most recent write event of x; if $i = j$*

and e_i^k is a relevant write of x then this number also includes e_i^k. Finally and most importantly, $\mathcal{A}$ should correctly implement the relative causality (stated formally in Theorem 3).

We next show how such an algorithm can be derived from its requirements above. In order to do it, let us first introduce some useful formal notation. For an event e_i^k of thread t_i, let $(e_i^k]$ be the indexed set $\{(e_i^k]_j\}_{1\leq j\leq n}$, where $(e_i^k]_j$ is the set $\{e_j^l \mid e_j^l \in t_j,\ e_j^l \in \mathcal{R},\ e_j^l \prec e_i^k\}$ when $j \neq i$ and the set $\{e_i^l \mid l \leq k,\ e_i^l \in \mathcal{R}\}$ when $j = i$. Intuitively, $(e_i^k]$ contains all the events in the multithreaded computation that causally precede or are equal to e_i^k.

Lemma 1. *With the notation above, the following hold for all $1 \leq i, j \leq n$: (1) $(e_j^{l'}]_j \subseteq (e_j^l]_j$ if $l' \leq l$; (2) $(e_j^{l'}]_j \cup (e_j^l]_j = (e_j^{\max\{l',l\}}]_j$ for any l and l'; (3) $(e_j^l]_j \subseteq (e_i^k]_j$ for any $e_j^l \in (e_i^k]_j$; and (4) $(e_i^k]_j = (e_j^l]_j$ for some appropriate l.*

Proof: *(1)* is immediate, because for any $l' \leq l$, any event e_j^k at thread t_j preceding or equal to $e_j^{l'}$, that is one with $k \leq l'$, also precedes e_j^l.
(2) follows by *1.*, because it is either the case that $l' \leq l$, in which case $(e_j^{l'}]_j \subseteq (e_j^l]_j$, or $l \leq l'$, in which case $(e_j^l]_j \subseteq (e_j^{l'}]_j$. In either case *2.* holds trivially.
(3) There are two cases to analyze. If $i = j$ then $e_j^l \in (e_i^k]_j$ if and only if $l \leq k$, so *3.* becomes a special instance of *1.*. If $i \neq j$ then by the definition of $(e_i^k]_j$ it follows that $e_j^l \prec e_i^k$. Since $e_j^{l'} \prec e_j^l$ for all $l' < l$ and since $\prec$ is transitive, it follows readily that $(e_j^l]_j \subseteq (e_i^k]_j$.
(4) Since $(e_i^k]_j$ is a finite set of totally ordered events, it has a maximum element, say e_j^l. Hence, $(e_i^k]_j \subseteq (e_j^l]_j$. By *3.*, one also has $(e_j^l]_j \subseteq (e_i^k]_j$. □

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

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

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

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

It is obvious that $(e_i^k]_x^w \subseteq (e_i^k]_x^a$. Some or all of the thread-indexed sets of events above may be empty. By convention, if an event, say e, does not exist in $\mathcal{M}$, then we assume that the indexed sets $(e]$, $(e]_x^a$, and $(e]_x^w$ are all empty (rather than "undefined"). Note that if $\mathcal{A}$ is

implemented such that V_x^a and V_x^w store the corresponding numbers of elements in the index sets of $(e_i^k]_x^a$ and $(e_i^k]_x^w$ immediately after event e_i^k is processed by thread t_i, respectively, then *(b)* and *(c)* in the list of requirements for $\mathcal{A}$ are also fulfilled.

Even though the sets of events $(e_i^k]$, $(e_i^k]_x^a$ and $(e_i^k]_x^w$ have mathematically clean definitions, they are based on total knowledge of the multithreaded computation $\mathcal{M}$. Unfortunately, $\mathcal{M}$ can be very large in practice, so the computation of these sets may be inefficient. Since our analysis algorithms are *online*, we would like to calculate these sets *incrementally*, as the observer receives events from the running program. A key factor in devising efficient update algorithms is to find equivalent *recursive* definitions of these sets, telling us how to calculate a new set of events from similar sets that have been already calculated at previous event updates.

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

Lemma 2. *Let e_i^k be an event in $\mathcal{M}$ and let e_j^l be the event preceding it in $\mathcal{M}$. If e_i^k is – (1) an internal event then $(e_i^k] = (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}}$, $(e_i^k]_x^a = (e_j^l]_x^a$ for any $x \in S$, and $(e_i^k]_x^w = (e_j^l]_x^w$ for any $x \in S$; (2) a read of x event then $(e_i^k] = (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (e_j^l]_x^w$, $(e_i^k]_x^a = (e_i^k] \cup (e_j^l]_x^a$, $(e_i^k]_y^a = (e_j^l]_y^a$ for any $y \in S$ with $y \neq x$, and $(e_i^k]_z^w = (e_j^l]_z^w$ for any $z \in S$; and (3) a write of x event then $(e_i^k] = (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (e_j^l]_x^a$, $(e_i^k]_x^a = (e_i^k]$, $(e_i^k]_x^w = (e_i^k]$, $(e_i^k]_y^a = (e_j^l]_y^a$ for any $y \in S$ with $y \neq x$, and $(e_i^k]_y^w = (e_j^l]_y^w$ for any $y \in S$ with $y \neq x$.*

Proof: *(1)* For the first equality, first recall that $e_i^k \in (e_i^k]$ if and only if e_i^k is relevant. Therefore, it suffices to show that $e \prec e_i^k$ if and only if $e \prec e_i^{k-1}$ for any relevant event $e \in \mathcal{R}$. Since e_i^k is internal, it cannot be in relation $<_x$ with any other event for any shared variable $x \in S$, so by the definition of $\prec$, the only possibilities are that either e is some event $e_i^{k'}$ of thread t_i with $k' < k$, or otherwise there is such an event $e_i^{k'}$ of thread t_i with $k' < k$ such that $e \prec e_i^{k'}$. Hence, it is either the case that e is e_i^{k-1} (so e_i^{k-1} is also relevant) or otherwise $e \prec e_i^{k-1}$. In any of these cases, $e \in (e_i^{k-1}]$. The other two equalities are straightforward consequences of the definitions of $(e_i^k]_x^a$ and $(e_i^k]_x^w$.
(2) Like in the proof of *(1)*, $e_i^k \in (e_i^k]$ if and only if $e_i^k \in \mathcal{R}$, so it suffices to show that for any relevant event $e \in \mathcal{R}$, $e \prec e_i^k$ if and only if $e \in (e_i^{k-1}] \cup (e_j^l]_x^w$. Since e_i^k is a read of $x \in S$ event, by the definition of $\prec$ one of the following must hold: *(i)* $e = e_i^{k-1}$. In this case e_i^{k-1} is also relevant, so $e \in (e_i^{k-1}]$; *(ii)* $e \prec e_i^{k-1}$ – it is obvious in this case that $e \in (e_i^{k-1}]$; *(iii)* e is a write of x event and $e <_x e_i^k$ – in this case $e \in (e_j^l]_x^w$; *(iv)* there is some write of x event e' such that $e \prec e'$ and $e' <_x e_i^k$ – in this case $e \in (e_j^l]_x^w$, too. Therefore, $e \in (e_i^{k-1}]$ or $e \in (e_j^l]_x^a$.

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

(3) It suffices to show that for any relevant event $e \in \mathcal{R}$, $e \prec e_i^k$ if and only if $e \in (e_i^{k-1}] \cup (e_j^l]_x^a$. Since e_i^k is a write of $x \in S$ event, by the definition of $\prec$ one of the following must hold: *(i)* $e = e_i^{k-1}$ – in this case $e_i^{k-1} \in \mathcal{R}$, so $e \in (e_i^{k-1}]$; *(ii)* $e \prec e_i^{k-1}$ – it is obvious in this case that $e \in (e_i^{k-1}]$; *(iii)* e is an access of x event (read or write) and $e <_x e_i^k$ – in this case $e \in (e_j^l]_x^a$; *(iv)* there is some access of x event e' such that $e \prec e'$ and $e' <_x e_i^k$ – in this case $e \in (e_j^l]_x^a$, too. Therefore, $e \in (e_i^{k-1}]$ or $e \in (e_j^l]_x^a$. For the second equality, note that, as for the second equation in *(2)*, one can readily see that $e \in (e_i^k]_x^a$ if and only if $e \in (e_i^k] \cup (e_j^l]_x^a$. But $(e_j^l]_x^a \subseteq (e_i^k]$, so the above is equivalent to $e \in (e_i^k]$. A similar reasoning leads to $(e_i^k]_x^w = (e_i^k]$. The equalities for $y \neq x$ immediate, because e_i^k has no relation to accesses of other shared variables but x. □

Since each component set of each of the indexed sets in these recurrences has the form $(e_i^k]_i$ for appropriate i and k, and since each $(e_i^k]_i$ can be safely encoded by its size, one can then safely encode each of the above indexed sets by an n-dimensional MVC; these MVCs are precisely V_i for all $1 \leq i \leq n$ and V_x^a and V_x^w for all $x \in S$. It is a simple exercise now to derive the following MVC update algorithm $\mathcal{A}$ (which was also given in Section 1):

ALGORITHM $\mathcal{A}$
INPUT: event e generated by thread t_i
1. if e is relevant then $V_i[i] \leftarrow V_i[i] + 1$
2. if e is a read of a shared 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 is a write of a shared variable x then $V_x^w \leftarrow V_x^a \leftarrow V_i \leftarrow \max\{V_x^a, V_i\}$
4. if e is relevant then send message $\langle e, i, V_i \rangle$ to observer

An interesting observation is that one can regard the problem of recursively calculating $(e_i^k]$ as a dynamic programming problem. As can often be done in dynamic programming problems, one can reuse space and derive the Algorithm $\mathcal{A}$. Therefore, $\mathcal{A}$ satisfies all the stated requirements (a), (b) and (c), so they can be used as properties next in order to show the correctness of $\mathcal{A}$:

Theorem 3. *If $\langle e, i, V \rangle$ and $\langle e', i', V' \rangle$ are two messages sent by $\mathcal{A}$, then $e \triangleleft e'$ if and only if $V[i] \leq V'[i]$ (no typo: the second i is not an i') if and only if $V < V'$.*

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

Synchronization and shared variables. Thread communication was considered so far to be accomplished by writing/reading shared variables, which were assumed to be known *a priori*. In the context of a language like Java, this assumption works only if the shared variables are declared *static*; it is less intuitive when synchronization and dynamically shared variables are considered as well. Here we claim that, under proper instrumentation, the basic algorithm presented in the previous subsection also works in the context of synchronization statements

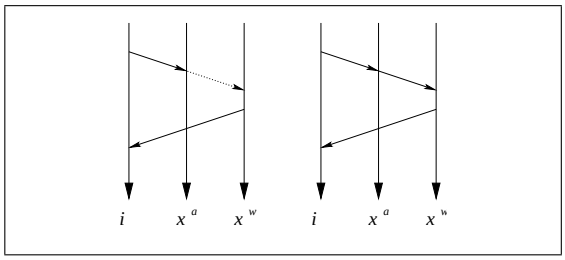

Figure 2. A distributed systems interpretation of reads (left) and writes (right).

and dynamically shared variables. Since in Java code blocks synchronized on same lock 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.

Distributed systems interpretation. It is known that the various mechanisms for process interaction are essentially equivalent. This leads to the question whether it could be possible to derive the MVC algorithm from vector clock based algorithms implementing causality in distributed systems, such as the ones in [2, 7]. Since writes and accesses of shared variables have different impacts on the causality relation, the most natural thing to do is to associate two processes to each shared variable x, one for accesses, say x^a and one for writes, say x^w. As shown in Fig. 2 right, a write of x by thread i can be seen as sending a "request" message to write x to the "access process" x^a, which further sends a "request" message to the "write process" x^w, which performs the action and then sends an acknowledgment messages back to i. This is consistent with step 3 of our algorithm; to see this, note that $V_x^w \leq V_x^a$ at any time.

However, a read of x is less obvious and does not seem to be interpretable by message passing the standard way. The problem here is that the MVC of x^a needs to be updated with the MVC of the accessing thread i, the MVC of the accessing thread i needs to be updated with the current MVC of x^w to implant causal dependencies between previous writes of x and the current access, but the point here is that the MVC of x^w does *not* have to be updated by reads of x; this is what allows reads to be permutable by the observer. In terms of message passing, like Fig. 2 shows, this says that the access process x^a sends a *hidden* request message to x_w (after receiving the read request from i), whose only role is to "ask" x^w send an acknowledgment to i. By hidden message, marked dotted in Fig. 2, we mean a message which is not considered by the standard MVC update algorithm. The role of the acknowledgment message is to ensure that i updates its MVC with the one of the write access process x^w.

4. The Vector Clock Algorithm at Work

Here we discuss predictive runtime analysis frameworks in which the presented MVC algorithm can be used, and describe how we use it in JAVA MULTIPATHEXPLORER (JMPAX) [25, 26, 18].

The observer therefore receives messages of the form $\langle e, i, V \rangle$ in any order, and, thanks to Theorem 3, can extract the causal partial order $\lhd$ on relevant events, which is its abstraction of the running program. Any permutation of the relevant events which is consistent with $\lhd$ is called a *multithreaded run*, or simply a *run*. Notice that each run corresponds to some possible execution of the program under different execution speeds or scheduling of threads, and that the observed sequence of events is just one such run. Since each relevant event contains global state update information, each run generates a sequence of global states. If one puts all these sequences together then one gets a lattice, called *computation lattice*. The reader is assumed familiar with techniques on how to extract a computation lattice from a causal order given by means of vector clocks [22]. Given a global property to analyze, the task of the observer now is to verify it against every path in the automatically extracted computation lattice. JPAX and JAVA-MAC are able to analyze only one path in the lattice. The power of our technique consists of its ability to predict potential errors in other possible multithreaded runs.

Once a computation lattice containing all possible runs is extracted, one can start using standard techniques on debugging distributed systems, considering both state predicates [27, 7, 5] and more complex, such as temporal, properties [3, 5, 1, 4]. Also, the presented algorithm can be used as a front-end to partial order trace analyzers such as POTA [24]. Also, since the computation lattice acts like an abstract model of the running program, one can potentially run one's favorite model checker against any property of interest. We think, however, that one can do better than that if one takes advantage of the specific runtime setting of the proposed approach. The problem is that the computation lattice can grow quite large, in which case storing it might become a significant matter. Since events are received incrementally from the instrumented program, one can buffer them at the observer's side and then build the lattice on a level-by-level basis in a top-down manner, as the events become available. The observer's analysis process can also be performed incrementally, so that parts of the lattice which become non-relevant for the property to check can be garbage-collected while the analysis process continues.

If the property to be checked can be translated into a finite state machine (FSM) or if one can synthesize online monitors for it, like we did for safety properties [26, 17, 17, 25], then one can analyze all the multithreaded runs *in parallel*, as the computation lattice is built. The idea is to store the state of the FSM or of the synthesized monitor together with each global state in the computation lattice. This way, in any global state, all the information needed about the past can be stored via a set of states in the FSM or the monitor associated to the property to check, which is typically quite small in comparison to the computation lattice. Thus only one cut in the computation lattice is needed at any time, in particular one level, which significantly reduces the space required by the proposed predictive analysis algorithm.

Java MultiPathExplorer (JMPaX) JMPAX [25, 26] is a runtime verification tool which checks a user defined specification against a running program. The specifications supported by JMPAX allow any temporal logic formula, using an interval-based notation built on state

predicates, so our properties can refer to the entire history of states. An instrumentation module parses the user specification, extracts the set of shared variables it refers to, i.e., the relevant variables, and then *instruments* the multithreaded program (which is assumed in bytecode form) as follows. Whenever a shared variable is accessed the MVC algorithm $\mathcal{A}$ in Section 3 is inserted; if the shared variable is relevant and the access is a write then the event is considered relevant. When the instrumented bytecode is executed, messages $\langle e, i, V\rangle$ for relevant events e are sent via a socket to an external observer.

The observer generates the computation lattice on a level by level basis, checking the user defined specification against all possible multithreaded runs in parallel. Note that only one of those runs was indeed executed by the instrumented multithreaded program, and that the observer does not know it; the other runs are *potential* runs, they can occur in other executions of the program. Despite the exponential number of potential runs, at most two consecutive levels in the computation lattice need to be stored at any moment. [25, 26] gives more details on the particular implementation of JMPaX. We next discuss two examples where JMPaX can predict safety violations from successful runs; the probability of detecting these bugs only by monitoring the observed run, as JPaX and Java-MAC do, is very low.

Example 1. Let us consider the simple landing controller in Fig.1, together with the property "If the plane has started landing, then it is the case that landing has been approved and since then the radio signal has never been down." Suppose that a successful execution is observed, in which the radio goes down *after* the landing has started. After instrumentation, this execution emits only three events to the observer in this order: a write of `approved` to 1, a write of `landing` to 1, and a write of `radio` to 0. The observer can now build the lattice in Fig.4, in which the states are encoded by triples `<landing,approved,radio>` and the leftmost path corresponds to the observed execution. However, the lattice contains two other runs both violating the safety property. The rightmost one corresponds to the situation when the radio goes down right between the test `radio==0` and the action `approved=1`, and the inner one corresponds to that in which the radio goes down between the actions `approved=1` and `landing=1`. Both these erroneous behaviors are insightful and very hard to find by testing. JMPaX is able to build the two counterexamples very quickly, since there are only 6 states to analyze and three corresponding runs, so it is able to give useful feedback.

Example 2. Let us now consider an artificial example intended to further clarify the prediction technique. Suppose that one wants to monitor the safety property "if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false" against a multithreaded program in which initially $x = -1$, $y = 0$ and $z = 0$, with one thread containing the code $x{+}{+};\ ...;\ y = x + 1$ and another containing $z = x + 1;\ ...;\ x{+}{+}$. The dots indicate code that is not relevant, i.e., that does not access the variables x, y and z. This multithreaded program, after instrumentation, sends messages to JMPaX's observer whenever the relevant variables x, y, z are updated. A possible execution of the program to be sent to the observer can consist of the sequence of program states $(-1, 0, 0)$, $(0, 0, 0)$, $(0, 0, 1)$, $(1, 0, 1)$, $(1, 1, 1)$, where the tuple $(-1, 0, 0)$ denotes the state in which $x = -1, y = 0, z = 0$. Following the MVC algorithm, we can deduce that the observer will receive the multithreaded computation shown in Fig. 3, which generates the computation lattice shown in the same figure. Notice that the observed multithreaded execution corresponds to just one particular multithreaded run out

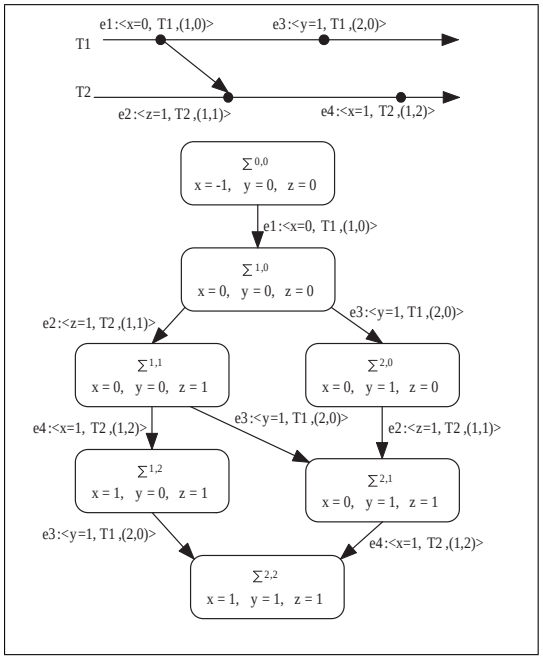

Figure 3. Computation lattice with three runs.

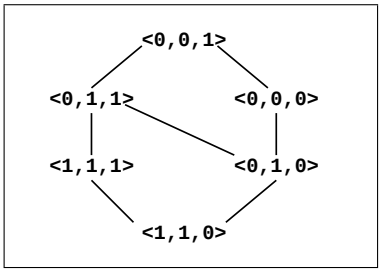

Figure 4. Computation lattice for the program in Figure 1.

of the three possible, namely the leftmost one. However, another possible run of the same computation is the rightmost one, which violates the safety property. Systems like JPaX and Java-MaC that analyze only the observed runs fail to detect this violation. JMPaX predicts this bug from the original successful run.

5. Conclusion

An algorithm for extracting the relevant causal dependency relation from a running multithreaded program was presented in this paper. This algorithm is supported by JMPaX, a runtime verification tool able to detect and predict safety errors in multithreaded programs.

Acknowledgments. Many thanks to Gul Agha and Mark-Oliver Stehr for their comments on previous drafts of this work. The work is supported in part by the ONR Grant N00014-02-1-0715 and the joint NSF/NASA grant CCR-0234524.

REFERENCES

1. M. Ahamad, M. Raynal, and G. Thia-Kime. An adaptive protocol for implementing causally consistent distributed services. In *Proc. of International Conference on Distributed Computing (ICDCS'98)*, pages 86–93, 1998.
2. O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.
3. O. Babaoğlu and M. Raynal. Specification and verification of dynamic properties in distributed computations. *Journal of Parallel and Distr. Computing*, 28(2):173–185, 1995.
4. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proc. Verification, Model Checking and Abstract Interpretation (VMCAI 04)*, volume 2937 of *LNCS*, pages 44–57, Jan. 2004.
5. C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.
6. E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.
7. R. Cooper and K. Marzullo. Consistent detection of global predicates. *ACM SIGPLAN Notices*, 26(12):167–174, 1991.
8. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proc. of Formal Methods Europe (FME'93): Industrial Strength Formal Methods*, volume 670 of *LNCS*, pages 268–284, 1993.
9. D. Drusinsky. Temporal rover. http://www.time-rover.com.
10. D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proc. SPIN Model Checking and Software Verification (SPIN'00)*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
11. D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *Proc. Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 114–118. Springer, 2003.
12. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 2002.
13. C. J. Fidge. Partial orders for parallel debugging. In *Proc. of the 1988 ACM SIGPLAN/SIGOPS workshop on Parallel and Distr. Debugging*, pages 183–194. ACM, 1988.
14. 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. Springer, 2003.
15. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proc. of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier, 2001.
16. K. Havelund and G. Roşu. *Runtime Verification 2001, 2002*, volume 55, 70(4) of *ENTCS*. Elsevier, 2001, 2002. Proc. of a *Computer Aided Verification (CAV'01, CAV'02)* workshop.
17. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools and Tech. Transfer*, 6(2):158–173, 2004.
18. Java MultiPathExplorer. `http://fsl.cs.uiuc.edu/jmpax`.
19. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proc. of Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier Science, 2001.
20. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proc. of the IEEE*, volume 84, pages 1090–1126, 1996.
21. K. Marzullo and G. Neiger. Detection of global state predicates. In *Proc. of the 5th International Workshop on Distributed Algorithms (WADG'91)*, volume 579 of *LNCS*, pages 254–272. Springer, 1991.
22. 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.
23. G. Roşu and K. Sen. An instrumentation technique for online analysis of multithreaded programs. In *PADTAD workshop at IPDPS*. IEEE Computer Society, 2003.
24. A. Sen and V. K. .Garg. Partial order trace analyzer (pota) for distrubted programs. In *Proc. of Workshop on Runtime Verification (RV'03)*, ENTCS, 2003.
25. K. Sen, G. Roşu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *9th European Software Engineering Conference and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*, pages 337–346. ACM, 2003.
26. K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 123–138, Barcelona, Spain, Mar. 2004. Springer.
27. S. D. Stoller. Detecting global predicates in distributed systems with clocks. In *Proc. of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, pages 185–199, 1997.
28. S. A. Vilkomir and J. P. Bowen. Formalization of software testing criteria using the Z notation. In *Proc. of 25th IEEE Annual International Computer Software and Applications Conference (COMPSAC'01)*, pages 351–356. IEEE Computer Society, Oct. 2001.