An Instrumentation Technique for Online Analysis of Multithreaded Programs

Grigore Roşu and Koushik Sen
Department of Computer Science,
University of Illinois at Urbana-Champaign, USA
Email: {grosu,ksen}@uiuc.edu

Abstract

A formal analysis technique aiming at finding safety errors in multithreaded systems at runtime *is investigated. An automatic code instrumentation procedure based on* multithreaded vector clocks *for generating the causal partial order on relevant state update events from a running multithreaded program is first presented. Then, by means of several examples, it is shown how this technique can be 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 is successful. The proposed technique has been implemented as part of a Java program analysis tool. A bytecode instrumentation package is used, so the Java source code of the tested programs is not necessary.*

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 exposed. To ameliorate this problem, test-case generation techniques have been developed to generate those test cases that can reveal potential errors with high probability [8, 18, 26]. Based on experience with and on the success in practice of related techniques already implemented in JAVA PATHEXPLORER (JPAX) [12, 11] and its sub-system EAGLE [4], we have proposed in [23, 24] a complementary approach to testing, which we call "predictive runtime analysis" and can be intuitively described as follows.

Suppose that a multithreaded program has a subtle safety error. 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 [12, 11], University of Pennsylvania's JAVA-MAC [17], or Bell Labs' PET [10], then there is not much left to do to find the error except to run another 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* [13], a more scalable and complementary approach to the traditional formal verification methods such as theorem proving and model checking [6]. Our focus here is on multithreaded systems with shared variables. More precisely, 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 properly *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) [23, 24], 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 shared 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 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 (the entire counterexample execution) 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 [9, 21], called *multithreaded vector clock (MVC)*. 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 code in Fig. 2 is executed. We prove that $\mathcal{A}$ correctly implements the relevant causal partial order, i.e., that for any two messages $\langle e, i, V\rangle$ and $\langle e', j, V'\rangle$ sent by $\mathcal{A}$, e and e' are relevant and e causally precedes e' iff $V[i] \leq V'[i]$. 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. All these can add significant delays to the normal execution of programs.

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

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

Figure 1. A buggy implementation of a flight controller.

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

Figure 2. The vector clock instrumentation algorithm.

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 algorithm which, given an executing 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.

2.1. 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^j to represent the j-th event generated by thread t_i since the start of its execution. When the thread or position of an event is not important we can refer to it generically, such as e, e', etc.; we may write $e \in t_i$ when event e is generated by thread t_i. Let us fix an arbitrary but fixed multithreaded execution, say $\mathcal{M}$, and let S be the set of all shared variables. There is an immediate notion of *variable access precedence* for each shared variable $x \in S$: we say e *x-precedes* e', written $e <_x e'$, 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 realized in practice by keeping a counter for each shared variable, which is incremented at each variable access.

2.2. Causality and Multithreaded Computations

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

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

We write $e||e'$ if $e \not\prec e'$ and $e' \not\prec e$. The partial order $\prec$ on $\mathcal{E}$ defined above is called the *multithreaded computation* associated with the original multithreaded execution $\mathcal{M}$. Synchronization of threads can be easily and elegantly taken into consideration by just generating 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 permutation of all events e_1, e_2, ..., e_r that does not violate the multithreaded computation, in the sense that the order of events in the permutation is consistent with $\prec$, is called a *consistent multithreaded run*, or simply, a *multithreaded run*.

A multithreaded computation can be thought of as the *most general assumption* that an observer of the multithreaded execution can make about the system without knowing its semantics. Indeed, an external observer simply *cannot disregard* the order in which the same variable is modified and used within the observed execution, because this order can be part of the intrinsic semantics of the multithreaded program. However, multiple causally independent modifications of different variable 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 [12, 11], JAVA-MAC [17], and PET [10], one gets the benefit of not only properly dealing with potential reorderings of delivered messages (e.g., due to using multiple channels to reduce the monitoring overhead), but also of *predicting errors* from analyzing successful executions, errors which can occur under a different thread scheduling.

2.3. Relevant Causality

Some variables in S may be of no importance for an external observer. For example, consider an observer whose purpose is to check the property "if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false"; formally, using the interval temporal logic notation notation in [15], 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 [20] 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 $e \prec e'$. It is important to notice though that the other variables can also indirectly influence the relation $\lhd$, because they can influence the relation $\prec$.

3. Multithreaded Vector Clock Algorithm

In this section, inspired and stimulated by the elegance and naturality of vector clocks [9, 21, 3] 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, after several unsuccessful attempts to design it on a less rigorous basis. In this section we present it also in a mathematically driven style, because we believe that it reflects an instructive methodology to devise instrumentation algorithms for multithreaded systems.

Let V_i be an n-dimensional vector of natural numbers for each $1 \leq i \leq n$. Since communication in multithreaded systems is done via shared variables, and since reads and writes have different weights, 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, $V \leq V'$ iff $V[j] \leq V'[j]$ for all $1 \leq j \leq n$, and $V < V'$ iff $V \leq V'$ and there is some $1 \leq j \leq n$ such that $V[j] < V'[j]$; also, $\max\{V, V'\}$ is 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 extract the relevant causal dependency relation. Formally, the requirements of such a procedure, say $\mathcal{A}$, which works as a filter of the given multithreaded execution, must include the following natural

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*[1] *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).

In order to derive our algorithm $\mathcal{A}$ satisfying the properties above, let us first introduce some 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 $j = i$.

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

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.*

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 $\mathcal{A}$ would be fulfilled.

Let us next move to the MVCs of reads and writes of shared variables. For a variable $x \in S$, let $a_x(e_i^k)$ and $w_x(e_i^k)$ be, respectively, the most recent events that accessed x and wrote x in $\mathcal{M}$, respectively. If such events do not exist then we let $a_x(e_i^k)$ and/or $w_x(e_i^k)$ undefined; if e is undefined then we also assume that $(e]$ is empty. We introduce the following notations for any $x \in S$:

$$(e_i^k]_x^a = \begin{cases} (e_i^k] & \text{if } e_i^k \text{ is an access to } x \text{, and} \\ (a_x(e_i^k)] & \text{otherwise;} \end{cases}$$

$$(e_i^k]_x^w = \begin{cases} (e_i^k] & \text{if } e_i^k \text{ is a write to } x \text{, and} \\ (w_x(e_i^k)] & \text{otherwise.} \end{cases}$$

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.

We next focus on how MVCs need to be updated by $\mathcal{A}$ when event e_i^k is encountered. With the notation introduced, one can observe the following recursive properties, where $\{e_i^k\}_i^{\mathcal{R}}$ is the indexed set whose 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:

Lemma 2 *Given any event e_i^k in $\mathcal{M}$ such that e_i^k is*

1. *An internal event then*

$$\begin{aligned} (e_i^k] &= (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}}, \\ (e_i^k]_x^a &= (a_x(e_i^k)], \textit{ for any } x \in S, \\ (e_i^k]_x^w &= (w_x(e_i^k)], \textit{ for any } x \in S; \end{aligned}$$

2. *A read of x event then*

$$\begin{aligned} (e_i^k] &= (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (w_x(e_i^k)], \\ (e_i^k]_x^a &= (e_i^k] \cup (a_x(e_i^k)], \\ (e_i^k]_y^a &= (a_y(e_i^k)], \textit{ for any } y \in S \textit{ with } y \neq x, \\ (e_i^k]_y^w &= (w_y(e_i^k)], \textit{ for any } y \in S; \end{aligned}$$

3. *A write of x event then*

$$\begin{aligned} (e_i^k] &= (e_i^{k-1}] \cup \{e_i^k\}_i^{\mathcal{R}} \cup (a_x(e_i^k)], \\ (e_i^k]_x^a &= (e_i^k], \\ (e_i^k]_x^w &= (e_i^k], \\ (e_i^k]_y^a &= (a_y(e_i^k)], \textit{ for any } y \in S \textit{ with } y \neq x, \\ (e_i^k]_y^w &= (w_y(e_i^k)], \textit{ for any } y \in S \textit{ with } y \neq x. \end{aligned}$$

Since each component set of each of the indexed sets in these recurrences has the form $(e_j^l]_j$ for appropriate j and l, and since each $(e_j^l]_j$ 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[2] the MVC update algorithm $\mathcal{A}$ given in Section 1. Therefore, $\mathcal{A}$ satisfies all the stated requirements (a), (b) and (c), so they can be used as properties next:

[1]Most recent with respect to the given multithreaded execution $\mathcal{M}$.

[2]An interesting observation here 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 $\mathcal{A}$.

Theorem 3 *If $\langle e, i, V\rangle$ and $\langle e', j, V'\rangle$ are two messages sent by $\mathcal{A}$, then $e \lhd e'$ if and only if $V[i] \leq V'[i]$ if and only if $V < V'$.*

Proof: First, note that e and e' are both relevant. The case $i = j$ is trivial. Suppose $i \neq j$. 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 $e \lhd e'$ 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 j such that e was generated by t_i and e' by t_j, and since $e' \not\lhd e$, by the first part of the theorem it follows that $V'[j] > V[j]$; therefore, $V < V'$. For the other implication, if $V < V'$ then $V[i] \leq V'[i]$, so the result follows by the first part of the theorem. $\square$

3.1. Synchronization and Shared Variables

Thread communication in multithreaded systems 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 show that, under proper instrumentation, the basic algorithm presented in the previous subsection also works in the context of synchronization statements and dynamically shared variables.

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

To handle variables that are dynamically shared, for each variable `x` of primitive type in each class the instrumentation program adds *access* and *write* MVCs, namely `_access_mvc_x` and `_write_mvc_x`, as new fields in the class. Moreover, for each read and write access of a variable of primitive type in any class, it adds codes to update the MVCs according to the multithreaded vector clock algorithm.

3.2. A Distributed Systems Interpretation

It is known that the various mechanisms for process interaction are essentially equivalent. This leads to the following natural question: could it be possible to derive the MVC algorithm in this section from vector clock based algorithms implementing causality in distributed systems, such as the ones in [3, 7]. The answer to this question is: *almost*.

Since writes and accesses of shared variables have different impacts on the causal dependency 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. 3 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 the algorithm in Fig. 2; 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 updating the MVCs 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 in order 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. 3 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 message to i. By hidden message, marked with dotted line in Fig. 3, 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.

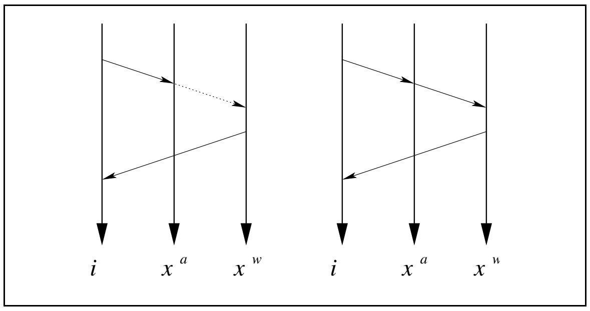

Figure 3. A distributed systems interpretation of reads (left) and writes (right).

4. The Vector Clock Algorithm at Work

In this section we propose predictive runtime analysis frameworks in which the presented MVC algorithm can be used, and describe by examples how we use it in JAVA MULTIPATHEXPLORER (JMPAX) [23, 24, 16].

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 [21]. 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 [25, 7, 5] and more complex, such as temporal, properties [2, 5, 1, 4]. Also, the presented algorithm can be used as a front-end to partial order trace analyzers such as POTA [22]. 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 [24, 14, 15, 23], 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.

Liveness properties apparently do not fit our runtime verification setting. However, stimulated by recent encouraging results in [19], we believe that it is also worth exploring techniques that can *predict violations of liveness properties*. The idea here is to search for paths of the form uv in the computation lattice with the property that the shared variable global state of the multithreaded program reached by u is the same as the one reached by uv, and then to check whether uv^ω satisfies the liveness property. The intuition here is that the system can potentially run into the infinite sequence of states uv^ω (u followed by infinity many repetitions of v), which may violate the liveness property. It is shown in [19] that the test $uv^\omega \models \varphi$ can be done in polynomial time and space in the sizes of u, v and φ, typically linear in uv, for almost any temporal logic.

4.1. Java MultiPathExplorer (JMPaX)

JMPAX [23, 24] 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. Fig. 4 shows the architecture of JMPAX. An instrumentation module parses the user specifica-

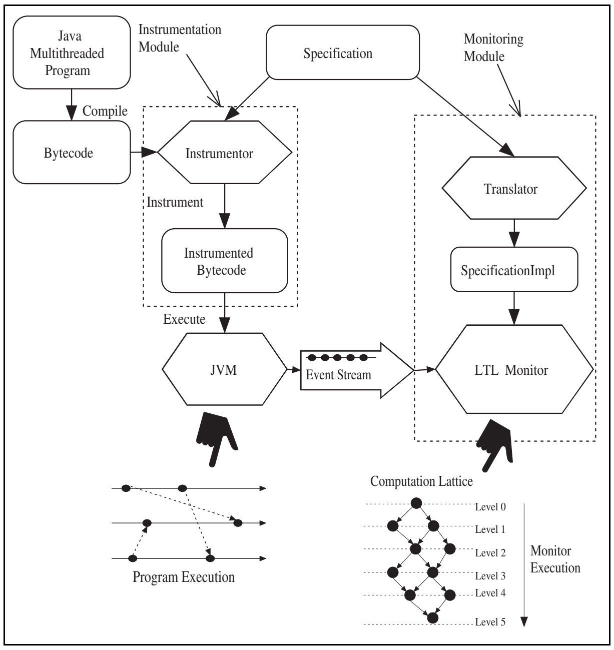

Figure 4. The Architecture of JMPAX.

tion, 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. [23, 24] 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.5, 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.

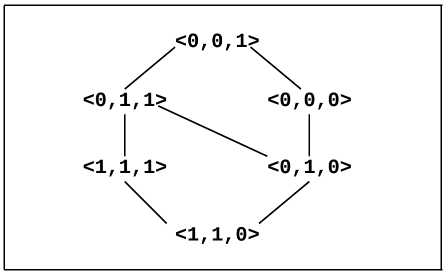

Figure 5. Computation lattice for the program in Fig. 1.

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. 6, which generates the computation lattice shown in the same figure. Notice that the observed multithreaded execution corresponds to just one particular multithreaded run out 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.

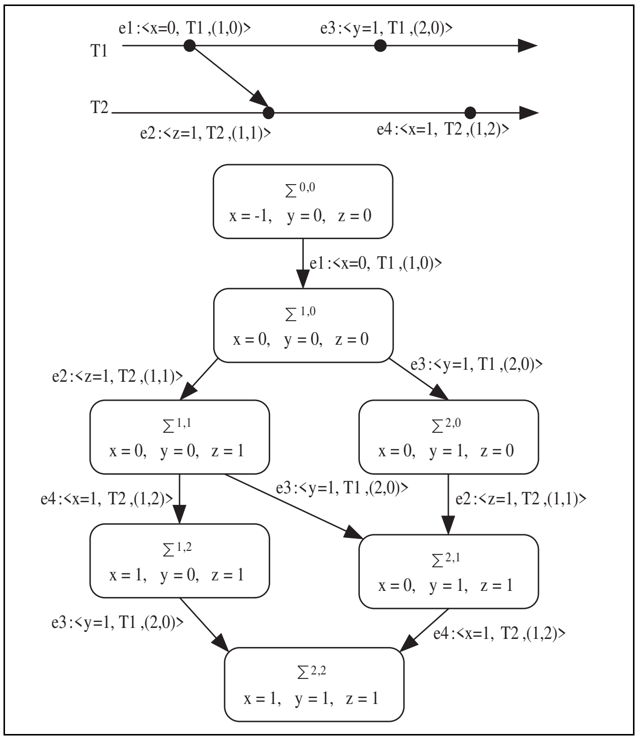

Figure 6. Computation lattice with three runs.

5. Conclusion

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

References

[1] M. Ahamad, M. Raynal, and G. Thia-Kime. An adaptive protocol for implementing causally consistent distributed services. In *Proceedings of International Conference on Distributed Computing (ICDCS'98)*, pages 86–93, 1998.

[2] O. Babaoglu and M. Raynal. Specification and verification of dynamic properties in distributed computations. *Journal of Parallel and Distr. Computing*, 28(2):173–185, 1995.

[3] 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.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings Verification, Model Checking and Abstract Interpretation (VMCAI 04) (To appear in LNCS)*, January 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. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.

[8] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of Formal Methods Europe (FME'93): Industrial Strength Formal Methods*, volume 670 of *LNCS*, pages 268–284, 1993.

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

[10] E. L. Gunter, R. P. Kurshan, and D. Peled. PET: An interactive software testing tool. In *Computer Aided Verification (CAV'00)*, volume 1885 of *LNCS*, pages 552–556. Springer-Verlag, 2003.

[11] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier, 2001.

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

[13] K. Havelund and G. Roşu. *Runtime Verification 2001, 2002*, volume 55, 70(4) of *ENTCS*. Elsevier, 2001, 2002. Proceedings of a *Computer Aided Verification (CAV'01, CAV'02)* workshop.

[14] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools and Tech. Transfer*, to appear.

[15] 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. Springer, 2002.

[16] Java MultiPathExplorer. `http://fsl.cs.uiuc.edu/jmpax`.

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

[18] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[19] N. Markey and P. Schnoebelen. Model checking a path (preliminary report). In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR'2003)*, LNCS. Springer, 2003.

[20] K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WADG'91)*, volume 579 of *LNCS*, pages 254–272. Springer, 1991.

[21] 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.

[22] A. Sen and V. K. .Garg. Partial order trace analyzer (pota) for distrubted programs. In *Proceedings of Workshop on Runtime Verification (RV'03)*, ENTCS, 2003.

[23] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*. ACM, 2003.

[24] 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) (To Appear in LNCS)*, Bercelona, Spain, 2004. Springer.

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

[26] S. A. Vilkomir and J. P. Bowen. Formalization of software testing criteria using the Z notation. In *Proceedings of COMPSAC 01: 25th IEEE Annual International Computer Software and Applications Conference*, pages 351–356. IEEE Computer Society, Oct. 2001.