PREDICTIVE SAFETY ANALYSIS OF CONCURRENT PROGRAMS

BY

KOUSHIK SEN

B.Tech., Indian Institute of Technology at Kanpur, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

Acknowledgments

The work is supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586 and the DARPA IXO NEST Program, contract number F33615-01-C-1907) and ONR Grant N00014-02-1-0715.

I would like to thank my advisor, Prof. Gul Agha, for his advice, support and intellectual guidance. He has always been available and willing to discuss anything.

A special thanks to Prof. Grigore Roşu for getting me started in this new research area and discussing the details of the technique.

I gratefully thank Abhay Vardhan for reviewing earlier drafts of the thesis. I would also like to thank Prasanna Thati, Wooyoung Kim for having discussion on the ideas presented in this thesis.

I would like to thank my wife and my parents for creating a healthy environment for work and for providing emotional support in times of crisis.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Systems in the real world consist many of distributed components that interact with each other concurrently. Such systems are called *concurrent systems*. Concurrent systems are implemented in different ways in practice, the most common paradigms being *message passing systems* and *shared memory systems*. In message passing systems, the different components, which can be threads, processes, or *actors* [2], communicate by exchanging messages. The mode of message exchange can be asynchronous or synchronous. In shared memory systems, the different components of the system communicate through updates of shared variables in a common memory. Most modern programming languages support concurrent programming either through message passing or through shared memory. For example, Java supports concurrent programming through *threads*: threads communicate through reading and writing of shared variables. Programs consisting of multiple threads and shared variables are called *multithreaded* programs. In multithreaded programs, an execution can take many potential paths depending on the order and speed of execution of the different threads. Therefore the complexity of the system increases and ensuring the correctness of the system becomes difficult. Nevertheless, it may be very important to ensure that *something bad* never happens in the running system. Properties expressing the fact that "nothing bad happens in the system" are called *safety* properties. Checking safety properties in the running system is called *runtime safety analysis*. We present foundational, scalable techniques for runtime safety analysis of multithreaded programs. In this technique we assume that there is an entity, called *observer*, that performs the runtime safety analysis. We present a simple and effective algorithm that allows an external observer of an executing multithreaded program to detect and predict safety violation. The idea is to *instrument* the system before its execution, so that it will emit a sequence of events at runtime to an observer. The observer analyzes the sequence of events by considering all consistent reordering of the events and predicts potential safety violations in the system. A particularly appealing aspect of our approach is that, despite the fact that a single execution, or interleaving, of a concurrent program can be observed, a comprehensive analysis of *all possible executions* consistent with the observation is performed. Thus, tools built on our techniques have the ability to *predict* safety violation errors in concurrent programs by observing executions which themselves may be successful.

The work in this dissertation falls under the area that has recently been called *runtime verification* [13, 12]: a program is executed to see a property holds during its execution. Runtime safety analysis is a subarea that, in particular, analyzes safety properties in runtime. A major goal of runtime verification is to combine aspects of testing and formal methods techniques. Testing scales well, and is by far the most used technique in practice to validate software systems. On the other hand, one can explore the whole state space of the system and systematically and exhaustively to check if every possible computations done by a program satisfies the desired property. This technique is called *model checking*. However, in model checking the entire state space that must be explored may be very large and hence computationally intractable. By merging testing and formal specification as used in model checking, runtime verification aims to achieve the benefits of both approaches, while avoiding some of the pitfalls of *ad hoc* testing and the complexity of full-blown theorem proving and model checking. Of course, there is a price to be paid in order to obtain a scalable technique: the entire state space of the system cannot be covered. The suggested framework can only be used to examine single execution traces, and therefore cannot be used to *prove* a system correct. However, a single execution trace typically contains much more information than what appears at first sight. We show how one can analyze all the other concurrent executions that are hidden behind a particular observed execution.

Based on our algorithm, we have developed a prototype tool called *Java MultiPathExplorer* (JMPaX – see Figure 1.1). JMPaX can reveal errors in Java multithreaded programs that are hard or impossible to detect otherwise. Using a suitable logic, the user of JMPaX specifies safety properties of interest. The safety properties are formulas over the global state of a compiled multithreaded program. JMPaX calls an instrumentation script which automatically instruments the executable multithreaded program to emit relevant state update events to an external observer. Finally, the instrumented program is run on any JVM. During the execution, JMPaX analyzes the safety violation messages reported by the observer.

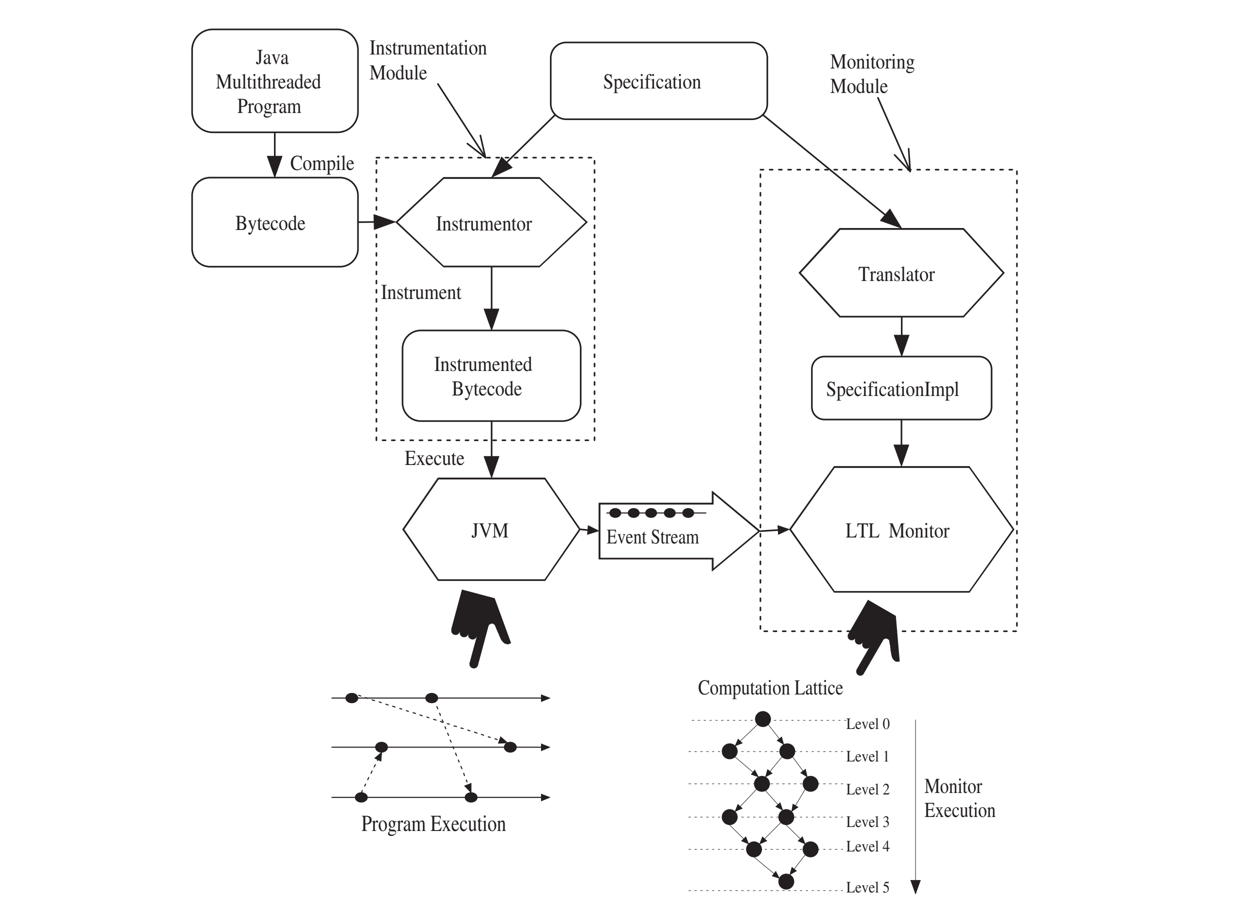

Figure 1.1: JMPaX Architecture

1.1 Example

To be more concrete, let us consider a real-life example where JMPaX was able to detect a violation of a safety property from a single execution of the program. However, the likelihood of detecting this bug only by monitoring an observed 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*, and 0 otherwise. The safety property that we want 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." As shown in Subsection 4.1, this property can be formally written in our extension of past time linear temporal logic as the formula

$$\uparrow\texttt{landing} \rightarrow [\texttt{approved}, \downarrow\texttt{radio})_s.$$

The code snippet for a naive implementation of this control program is given as follows:

```
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(){
   randomly change value of radio;}
```

The above code uses some dummy functions, namely `askLandingApproval` and `checkRadio`, which can be implemented in their entirety in a real scenario. The program has a serious problem which 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. 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 simple observer will probably not detect the bug. However, note that even if the radio goes off after the landing has started, JMPaX can still construct a possible run in which radio goes off between landing and approval. Thus JMPaX will be able to predict the safety violation from a single successful execution of the program. This example shows the power of our runtime verification technique.

1.2 Our Contributions

We can think of at least three major contributions of the work presented in this paper.

1. We nontrivially extend the runtime safety analysis capabilities of existing systems, by providing the ability to predict safety errors from successful executions in multithreaded systems; we are not aware of any other efforts in this direction.

2. We define the notion of relevant causality in multithreaded systems with shared variables and synchronization points; this notion is used to instrument multithreaded programs to emit to external observers a causal dependency partial relation on global state updates via relevant events timestamped with appropriate vector clocks.

3. We implemented a modular prototype runtime analysis system, JMPaX, showing that, despite their theoretical flavor, all the concepts presented in the paper are in fact quite practical and can lead to new scalable verification tools.

Chapter 2

Related Work

Our work combines aspect of the formalisms and techniques used commonly in model checking [8, 10], distributed debugging [21, 6, 5] and runtime-verification of safety properties [13, 12, 16, 17] to achieve an efficient, scalable, predictive technique for runtime verification of concurrent programs. We next discuss these commonly used techniques and compare them with our approach.

2.1 Model Checking

Model checking is a *push-button* technique for verifying finite state concurrent systems against required specification of the system. The tasks involved in model checking are as follows:

- A formal model of the system is given in terms of a state transition system. The state transition system is a tuple $\mathcal{M} = (S, S_0, R, L)$, where

  i. S is the finite set of states,

  ii. S_0 a subset of S is the set of initial states from which system can start its execution,

  iii. $R \subseteq S \times S$ is a total relation, describing the possible transitions from one state to another state of the system, and

iv. $L : S \rightarrow \mathcal{P}(AP)$ is a labelling function, stating the atomic propositions (AP) that hold in a given state.

The state transition system of a concurrent program can be constructed automatically by exploring all possible states of the system that can be reached from the initial states.

- The properties that the model must satisfy are stated as a specification. The specification is usually given in some logical formalism. The commonly used logics are temporal logics.

- After expressing the model and the formal specification, the verification task involves checking the conformance of the model to the given specification. In case of a negative result, a counter-example is generated. This task is completely automatic.

In model checking, all possible computations of the systems are analyzed. So the method is rigorous and complete. Model checking discovers a bug if it is present in the system. Theoretically, model-checking is very efficient. However, in practice model-checking may require the entire state space of the system to be stored before bug can be detected. This problem is called the *state space explosion* problem. In sequential programs input variables may have many possible values leading to a large number of possible states. In concurrent programs, nondeterministic execution can lead to a large number of states. If the total number of possible states of the system is large, model checking becomes intractable which makes this technique not scalable. We take the formal logics used to specify safety properties and incorporate the logics in our approach. This makes our approach more formal compared to the *ad hoc* testing used in traditional debugging.

2.2 Distributed Debugging

The other extreme approach for verification is distributed debugging [21, 6, 5]. The technique is *ad hoc* and involves the testing of certain state predicates on global states as the program is executed. In distributed debugging, distributed programs are assumed to be a set of processes communicating through asynchronous message passing. A monitor process is assumed to be responsible for debugging. An executing process sends events to the monitor whenever it changes its local state, sends a message to some other process, or receives a message from some other process. The monitor collects these events and constructs the sequence of global states through which the system passes. Because the message passing is asynchronous, messages can be received out of order. Thus, the global states constructed by the monitor from the sequence of events may be obsolete, incomplete, or inconsistent. A global state is inconsistent if it cannot be observed externally. The delay in message passing and difference in relative computation speed may prevent an external observer from constructing the global state of the system. Distributed debugging addresses this problem by using techniques to construct consistent global states and to evaluate state predicates on these states. The techniques use vector clocks to attach ordering information with the events. Each process maintains a vector clock, an n-dimensional vector of natural numbers, where n is the number of processes. Each entry V_i in the vector clock represents the latest event in the process i that have causally affected the current event in the process owning the vector clock. The vector clocks are updated whenever an event occurs in a process. These vector clocks are sent to the monitor along with the events. The monitor extracts information about the order of events from these vector clocks and constructs all possible consistent global states of the system. It then evaluates the state predicates on the consistent global states.

The use of vector clocks [9, 19] helps a monitor construct states that are consistent with the distributed computation but may not have occurred in the original execution. Thus, not only can a monitor *detect* bugs in the current execution, it can *predict* bugs in other possible consistent states of the system. Message passing systems are considered in distributed debugging. We take the concept of vector clock for message passing systems and extend it to multithreaded programs that communicate through shared variables.

2.3 Runtime Verification

A recent approach to scalable techniques for verification is to merge formal methods with traditional testing in the most obvious way, resulting in a new research area called runtime verification [12, 11, 13, 12, 16, 17]. The commonly cited works in this area are: a) NASA's PathExplorer (PaX) and its Java instance JPaX [12, 11], which is a runtime verification system developed at NASA Ames, and b) UPENN's MaC and its instance Java MaC [16, 17]. In runtime verification, the execution of the program is monitored as it is in the case of debugging. However, instead of monitoring simple state predicates, more sophisticated logics, such as past time temporal logic [14], linear temporal logic over finite traces [12] and extended regular expressions, are used to monitor temporal properties. The suggested frameworks can only be used to examine single execution traces, and therefore can not be used to *prove* a system correct. However, a single execution trace typically contains much more information than what appears at first sight. In this thesis, we show how one can analyze all other potential executions that are consistent with the observed execution and the information collected through vector clocks.

Chapter 3

Multithreaded Systems

We consider multithreaded systems in which several threads communicate with each other via a set of shared variables. A crucial point to note is that some of the variable updates can causally depend on others. We will present an algorithm which, given an executing multithreaded system, generates appropriate messages 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 causality partial order among the updates.

3.1 Multithreaded Executions and Shared Variables

Given n threads t_1, t_2, ..., t_n, a *multithreaded execution* is a sequence of events $e_1 e_2 \dots e_r$, 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.

3.2 Causality and Multithreaded Computations

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

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

We write $e||e'$ if $e \not\prec e'$ and $e' \not\prec e$. The 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 consecutive reads of the same 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 [11] and Java-MaC [16], one gets the benefit of not only properly dealing with potential reorderings of delivered messages (e.g., due to using multiple channels in order to reduce the monitoring overhead), but also of *predicting errors* from analyzing successful executions, errors which may occur under a different thread scheduling.

3.3 Relevant Causality

Some of the variables in S may be of no importance at all for an external observer. For example, consider an observer whose purpose is to check the property "if $(x > 0)$ then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false"; formally, using an interval temporal logic notation, this requirement can be compactly written as $(x > 0) \rightarrow [y = 0, y > z)_s$. All the other variables in S except x, y and z are essentially irrelevant for this observer. To minimize the number of messages 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 $\prec$. We next provide a technique based on *vector clocks* that correctly implements the relevant causality relation.

3.4 Deriving the Main Algorithm

Inspired and stimulated by the elegance and naturality of vector clocks in representing causal dependency in distributed systems [9, 19], we have devised a vector clock based algorithm to represent the relevant causal dependency relation in multithreaded systems. After several unsuccessful attempts to design it on a less rigorous basis, this algorithm was eventually mathematically derived from its desired properties. In this section we present the algorithm also in a mathematically driven style, because we believe that it reflects an instructive and methodology for devising vector clock based 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^a_x and V^w_x be two additional n-dimensional vectors for each shared variable x; we call the former *access vector clock* and the latter *write vector clock*. All vector clocks 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 an algorithm that updates these vector clocks and emits a minimal number of events to an external observer who can further extract the relevant causal dependency relation. The algorithm that we derive works as follows. Whenever a thread p_i with current vector clock V_i processes event e_i^k, the following vector clock algorithm is executed:

1. if e_i^k is relevant, i.e., if $e_i^k \in \mathcal{R}$, then

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

2. if e_i^k is a read of a variable x then

   $V_i \leftarrow \max\{V_i, V_x^w\}$

   $V_x^a \leftarrow \max\{V_x^a, V_i\}$

3. if e_i^k is a write of a variable x then

   $V_x^w \leftarrow V_x^a \leftarrow V_i \leftarrow \max\{V_x^a, V_i\}$

4. if e_i^k is relevant then

   send message $\langle e_i^k, i, V_i \rangle$ to observer.

Formally, the requirements of the above algorithm, say $\mathcal{A}$, which works as a filter of the given multithreaded execution, must include the following natural but crucial

Requirements for $\mathcal{A}$. *After $\mathcal{A}$ updates the vector clocks 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;

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

(c) $V_x^w[j]$ *equals the number of relevant events of* p_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 relation (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 $j = i$. The following is immediate:

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

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*.*

Therefore, 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 vector clock 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 vector clocks 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 $w_x(e_i^k)$ be undefined; if e is undefined then we also assume that $(e]$ is empty. We introduce the following notations for any $x \in S$:

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

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

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 these vector clocks need to be updated by $\mathcal{A}$ when an event e_i^k in encountered. With the notation introduced so far, 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*

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

Because each component set of each of the indexed sets occurring in the above recurrences is of the form $(e_j^l]_j$ for appropriate j and l, and because each $(e_j^l]_j$ can be safely encoded by a unique number, namely its size, one can then safely encode each of the above indexed sets by an n-dimensional vector clock; these vector clocks 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 vector clock update algorithm $\mathcal{A}$. Therefore, $\mathcal{A}$ satisfies all the stated requirements (a), (b) and (c), so they can be used as properties of $\mathcal{A}$ in the next theorem.

Theorem 3 *If $\langle e, i, V \rangle$ and $\langle e', j, V' \rangle$ are two messages sent by $\mathcal{A}$, then $e \vartriangleleft e'$ 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', then it is clear that $V[i] \leq V'[i]$ if and only if $e \prec e'$. For the second part, if $e \vartriangleleft e'$ 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 $e' \not\prec e$, 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. $\square$.

3.5 Computation Lattice

Consider what happens at the observer's site. The observer receives messages of the form $\langle e, i, V \rangle$ in any possible order. We let $\mathcal{R}$ denote the set of received relevant events, which we simply call *events* in what follows. By using Theorem 3, the observer can infer the causal

[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 $\mathcal{A}$.

dependency between the relevant events emitted by the multithreaded system. Inspired by similar definitions at the multithreaded system's [4], we define the important notions of relevant multithreaded computation and run as follows. A *relevant multithreaded computation*, simply called *multithreaded computation* from now on, is the partial order on events that the observer can infer, which is nothing but the relation $\triangleleft$. A *relevant multithreaded run*, also simply called *multithreaded run* from now on, is any permutation of the received events which *does not violate* the multithreaded computation. Our objective is to check safety requirements against *all* (relevant) multithreaded runs of a multithreaded system.

We assume that the relevant events are only writes of shared variables that appear in the safety formulae to be monitored, and that these events contain a pair of the name of the corresponding variable and the value which was written to it. We call these variables *relevant variables*. Note that events can change the state of the multithreaded system as seen by the observer; this is formalized next. A *relevant program state*, or simply a *program state* is a map from relevant variables to concrete values. Any permutation of events generates a sequence of program states in the obvious way, however, not all permutations of events are valid multithreaded runs. A program state is called *consistent* if and only if there is a multithreaded run containing that state in its sequence of generated program states. We next formalize these concepts.

For a given permutation of (relevant) events in $\mathcal{R}$, say $e_1 e_2 \ldots e_{|\mathcal{R}|}$, we can let e_i^k denote the k-th event of thread p_i for each $1 \leq i \leq n$. Then the relevant program state after the events $e_1^{k_1}, e_2^{k_2}, \ldots, e_n^{k_n}$ is called a *relevant global multithreaded state*, or simply a *relevant global state* or just *state*, and is denoted by $\Sigma^{k_1 k_2 \ldots k_n}$. A state $\Sigma^{k_1 k_2 \ldots k_n}$ is called *consistent* if and only if for any $1 \leq i \leq n$ and any $l_i \leq k_i$, it is the case that $l_j \leq k_j$ for any $1 \leq j \leq n$ and any l_j such that $e_j^{l_j} \triangleleft e_i^{l_i}$. Let Σ^{K_0} be the *initial* global state, $\Sigma^{00\ldots0}$. An important observation is that $e_1 e_2 \ldots e_{|\mathcal{R}|}$ is a multithreaded run if and only if it generates a sequence of global states $\Sigma^{K_0}\Sigma^{K_1}\ldots\Sigma^{K_{|\mathcal{R}|}}$ such that each Σ^{K_r} is consistent and for any two consecutive Σ^{K_r} and $\Sigma^{K_{r+1}}$, K_r and K_{r+1} differ in exactly one index, say i, where the i-th element in K_{r+1}

is larger by 1 than the i-th element in K_r. For that reason, we will identify the sequences of states $\Sigma^{K_0}\Sigma^{K_1}\ldots\Sigma^{K_{|\mathcal{R}|}}$ as above with multithreaded runs, and simply call them *runs*. We say that Σ *leads-to* Σ', written $\Sigma \leadsto \Sigma'$, when there is some run in which Σ and Σ' are consecutive states. The set of all consistent global states together with the relation $\leadsto$ forms a *lattice*. The lattice has n mutually orthogonal axis representing each thread. For a state $\Sigma^{k_1 k_2 \ldots k_n}$, we call $k_1 + k_1 + \cdots k_n$ its *level*. A *path* in the lattice is a sequence of consistent global states on increasing level, where the level increases by 1 between any two consecutive states in the path. Therefore, a run is just a path starting with $\Sigma^{00\ldots 0}$ and ending with $\Sigma^{r_1 r_2 \ldots r_n}$, where r_i is the total number of events of thread i for each $1 \leq i \leq n$. Therefore, a multithreaded computation can be seen as a lattice; we call this lattice a *computation lattice*.

Example 1. Suppose that one wants to monitor some safety property of the multithreaded program below. The program involves relevant variables x, y and z:

Initially: $x = -1; y = 0; z = 0;$

thread $T1\{$	*thread* $T2\{$
...	...
$x_{++};$	$z = x + 1;$
...	...
$y = x + 1;$	$x_{++};$
...	...
$\}$	$\}$

The ellipses indicate code that is not relevant, i.e., that does not access the variables x, y and z. This multithreaded program, after appropriate instrumentation, sends messages to an observer whenever the relevant variables are updated. A possible execution of the program to be sent to the observer, described in terms of relevant variable updates, can be

$$\{x = -1, y = 0, z = 0\}, \{x = 0\}, \{z = 1\}, \{y = 1\}, \{x = 1\}$$

The first message to observer sends the initial state of the whole system as a set of variable-value pairs. The second event is generated when the value of x is incremented by the first thread. The above execution corresponds to the sequence of program states

$$(-1, 0, 0),\ (0, 0, 0),\ (0, 0, 1),\ (0, 1, 1),\ (1, 1, 1)$$

where the tuple $(-1, 0, 0)$ denotes the state in which $x = -1, y = 0, z = 0$. Following the vector clock algorithm, we can deduce that the observer will receive the multithreaded computation in Figure 3.1 which generates the computation lattice shown in the same figure. Notice that the observed multithreaded execution corresponds to just one particular mul-

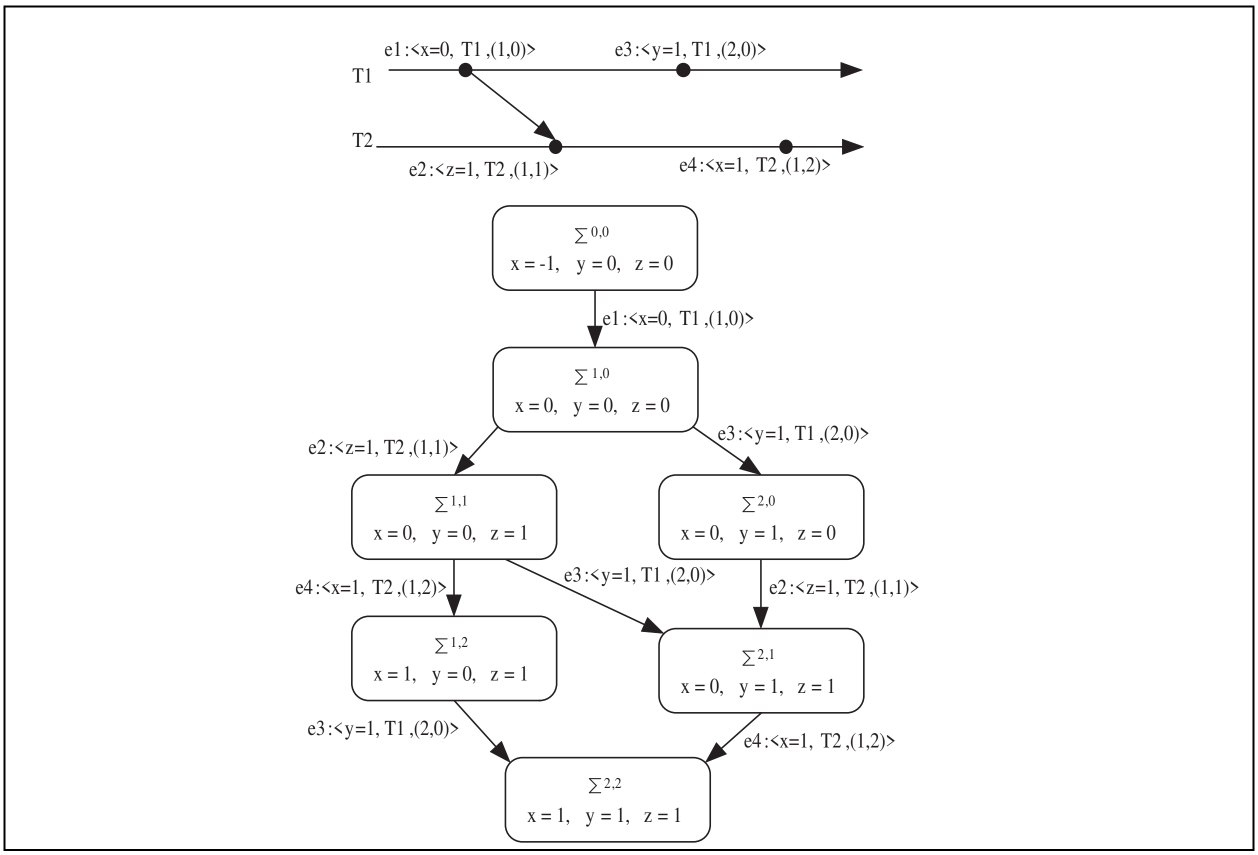

Figure 3.1: Computation lattice and three runs.

tithreaded run out of the three possible. We will show that it is often possible that the observed run does not violate any safety property, but the run nevertheless shows that there are other possible runs that are not safe. We will propose an algorithm that will detect safety violations in any possible run, even though the violation was not revealed by the particular observed run. An appealing aspect of our algorithm is that, despite the fact that there can be a potentially exponential number of runs (in the maximum width of a level), they can all be analyzed *in parallel*, by generating and traversing the lattice on a level-by-level basis; the previous levels are not needed, so memory can be reused.

Chapter 4

Multithreaded Safety Analysis

In this section, we first introduce the past time temporal logic that we use to express safety properties, then we recall an algorithm to monitor such properties efficiently against a single run, and finally we show how this algorithm nontrivially extends to monitoring multithreaded computations given as partial orders.

4.1 Safety in Temporal Logics

We use past time Linear Temporal Logic (*ptLTL*)[18] to express our safety properties. Our choice of past time linear temporal logic is motivated by two considerations:

1. It is powerful enough to express safety properties of concurrent systems;
2. The monitors for a safety formula written in *ptLTL* are very efficient; they perform linearly in the size of the formula in the worst case [14].

Now we briefly review the basic notions of *ptLTL* , and describe some new operators that are particularly useful for runtime monitoring. The syntax of *ptLTL* is given as follows:

$$\begin{array}{rll} F ::= & true \mid false \mid a \in A \mid \neg F \mid F\ op\ F & \text{Propositional ops} \\ & \odot F \mid \Diamond F \mid \boxdot F \mid F \mathcal{S}_s F \mid F \mathcal{S}_w F & \text{Standard operators} \\ & \uparrow F \mid \downarrow F \mid [F, F)_s \mid [F, F)_w & \text{Monitoring ops} \end{array}$$

$\rho \models true$	is true for all ρ,
$\rho \models a$	iff a holds in the state s_n,
$\rho \models \neg F$	iff $\rho \not\models F$,
$\rho \models F_1 \ op\ F_2$	iff $\rho \models F_1$ and/or/implies/iff $\rho \models F_2$, when op is $\wedge / \vee / \rightarrow / \leftrightarrow$,
$\rho \models \odot F$	iff $\rho' \models F$, where $\rho' = \rho_{n-1}$ if $n > 1$ and $\rho' = \rho$ if $n = 1$,
$\rho \models \Diamond F$	iff $\rho_i \models F$ for some $1 \leq i \leq n$,
$\rho \models \boxdot F$	iff $\rho_i \models F$ for all $1 \leq i \leq n$,
$\rho \models F_1 \mathcal{S}_s F_2$	iff $\rho_j \models F_2$ for some $1 \leq j \leq n$ and $\rho_i \models F_1$ for all $1 \leq i \leq n$,
$\rho \models F_1 \mathcal{S}_w F_2$	iff $\rho \models F_1 \mathcal{S}_s F_2$ or $\rho \models \boxdot F_1$,
$\rho \models \uparrow F$	iff $\rho \models F$ and it is not the case that $\rho \models \odot F$,
$\rho \models \downarrow F$	iff $\rho \models \odot F$ and it is not the case that $\rho \models F$,
$\rho \models [F_1, F_2)_s$	iff $\rho_j \models F_1$ for some $1 \leq j \leq n$ and $\rho_i \not\models F_2$ for all $j \leq i \leq n$,
$\rho \models [F_1, F_2)_w$	iff $[F_1, F_2)_s$ or $\rho \models \boxdot \neg F_2$,

Table 4.1: Semantics of *ptLTL*

where *op* are the standard binary operators, namely $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and $\odot F$ should be read as "previously", $\Diamond F$ as "eventually in the past", $\boxdot F$ as "always in the past", $F_1 \mathcal{S}_s F_2$ as "F_1 strong since F_2", $F_1 \mathcal{S}_w F_2$ as "F_1 weak since F_2", $\uparrow F$ as "start F"', $\downarrow F$ as "end F", $[F_1, F_2)_s$ as "strong interval F_1, F_2", and $[F_1, F_2)_w$ as "weak interval F_1, F_2".

The logic is interpreted on a finite sequence of states or a run. If $\rho = s_1 s_2 \ldots s_n$ is a run then we let ρ_i denote the prefix run $s_1 s_2 \ldots s_i$ for each $1 \leq i \leq n$. The semantics of the different operators is given in Table 4.1.

The monitoring operators $\uparrow$, $\downarrow$, $[,)_s$, and $[,)_w$ were introduced in [14, 17]. These operators have been found to be very intuitive and useful in specifying properties for runtime monitoring. Informally, $\uparrow F$ is true if and only if F *starts* to be true in the current state, $\downarrow F$ is true if and only if F *ends* being true in the current state, and $[F_1, F_2)_s$ is true if and only if F_2 was never true since the last time F_1 was observed to be true, including the state when F_1 was true; the interval operator has a strong and a weak version. For example, if *boot* and *down* are predicates on the state of a web server to be monitored, say for the last 24 hours, then $[boot, down)_s$ is a property stating that the server *was* rebooted recently and the since then it was not down, while $[boot, down)_w$ say that server was not unexpectedly down recently, meaning that it was either not down at all recently or it was rebooted recently and since then it was not down.

In runtime monitoring, we start the process of monitoring from the point the first event is generated and it continues as long as the events are generated. Thus given a *ptLTL* formula F we check whether $\rho_i \models F$ for all $1 \leq i \leq n$, where $\rho = s_1 s_2 \ldots s_n$ is a finite run constructed from the events.

4.2 Checking Safety against a Single Run

We describe an algorithm for monitoring the multithreaded execution or *the observed run* of a multithreaded computation, which is just one path in the computation lattice, and illustrate it through an example. This algorithm is a modified version of the algorithm presented in [14]. The algorithm computes the boolean value of the formula to be monitored, by recursively evaluating the boolean value of each of its subformulae in the current state. In the process, it also uses the boolean value of certain subformulae evaluated in the previous state. Next, we define this recursive function *eval*. The recursive nature of the temporal operators in *ptLTL* enables us to define the boolean value of a formula in the current state in terms of its boolean value in the previous state and the boolean value of its subformulae in the current state. For example we can define:

$$
\begin{array}{ll}
\rho \models \Diamond\!\!\cdot F & \text{iff } \rho \models F \text{ or } (n > 1 \text{ and } \rho_{n-1} \models \Diamond\!\!\cdot F) \\
\rho \models \boxdot F & \text{iff } \rho \models F \text{ and } (n > 1 \text{ implies } \rho_{n-1} \models \boxdot F) \\
\rho \models F_1 \mathcal{S}_s F_2 & \text{iff } \rho \models F_2 \text{ or} \\
 & (n > 1 \text{ and } \rho \models F_1 \text{ and } \rho_{n-1} \models F_1 \mathcal{S}_s F_2) \\
\rho \models F_1 \mathcal{S}_w F_2 & \text{iff } \rho \models F_2 \text{ or} \\
 & (\rho \models F_1 \text{ and } (n > 1 \text{ and } \rho_{n-1} \models F_1 \mathcal{S}_w F_2)) \\
\rho \models [F_1, F_2)_s & \text{iff } \rho \not\models F_2 \text{ and} \\
 & (\rho \models F_1 \text{ or } (n > 1 \text{ and } \rho_{n-1} \models [F_1, F_2)_s)) \\
\rho \models [F_1, F_2)_w & \text{iff } \rho \not\models F_2 \text{ and} \\
 & (\rho \models F_1 \text{ or } (n > 1 \text{ implies } \rho_{n-1} \models [F_1, F_2)_w))
\end{array}
$$

These definitions correspond to the code for the cases of the operators $\Diamond\!\!\cdot$, $\boxdot$, $\mathcal{S}_s$, $\mathcal{S}_w$, $[,)_s$, and $[,)_w$ in the function *eval*. The function *op(f)* returns the operator of the formula f, *binary(op(f))* returns *true* if *op(f)* is binary, *unary(op(f))* returns *true* if *op(f)* is true, *left(f)* returns the left subformula of f, *right(f)* returns the right subformula of f, and *subformula(f)* returns the subformula of f.

boolean

$eval(Formula\ f,$ **State** $s,$ **array** $now,$ **array** $pre,$ **int** $index)\{$

  if $binary(op(f))$ **then**$\{$

    $lval \leftarrow eval(left(f), now, pre, index);$

    $rval \leftarrow eval(right(f), now, pre, index);\}$

  else if $unary(op(f))$ **then**

    $val \leftarrow eval(subformula(f), now, pre, index);$

  case$(op(f))$ **of**$\{$

    p : **return** $p(s)$; $true$: **return** $true$; $false$: **return** $false$;

    op : **return** $rval\ op\ lval$; $\neg$: **return not** val;

    $\mathcal{S}_s, \mathcal{S}_w : now[{+}{+}index] \leftarrow (pre[index]$ **and** $lval)$ **or** $rval$;

return $(pre[index]$ **and** $lval)$ **or** $rval$;

$[,)_s, [,)_w$:

$now[{+}{+}index] \leftarrow$ (**not** $rval$) **and** ($pre[index]$ **or** $lval$);

return (**not** $rval$) **and** ($pre[index]$ **or** $lval$);

$\uparrow$: $now[{+}{+}index] \leftarrow val$;

return (**not** $pre[index]$) **and** val;

$\downarrow$: $now[{+}{+}index] \leftarrow val$;

return $pre[index]$ **and** (**not** val);

$\boxdot$: $now[{+}{+}index] \leftarrow pre[index]$**and** val; **return** $now[index]$;

$\Diamond$: $now[{+}{+}index] \leftarrow pre[index]$ **or** val; **return** $now[index]$;

$\odot$: $now[{+}{+}index] \leftarrow val$; **return** $pre[index]$;

}

}

Here, the *pre* array passed as an argument contains the boolean values of all subformulae in the previous state that will be required in the current state. The *now* array, after the evaluation of *eval* function, will contain the boolean values of all subformulae in the current state that will be required in the next state. Note, the *now* array is passed as reference, and its value is set in the function *eval*. The function *eval*, however, cannot be used to evaluate the boolean value of a formula for the first state in a run, as the recursion handles the case $n = 1$ in a different way. We define the function *init* to handle this special case as follows:

boolean $init(Formula\ f,$ **State** $s,$ **array** $now,$ **int** $index)\{$

if $binary(op(f))$ **then**{

$lval \leftarrow init(left(f), now, index)$;

$rval \leftarrow eval(right(f), now, index); \}$

else if $unary(op(f))$ **then**

$val \leftarrow init(subformula(f), now, index)$;

case$(op(f))$ **of**{

p : **return** $p(s)$; $true$: **return** $true$; $false$: **return** $false$;

op : **return** $rval$ op $lval$; $\neg$: **return not** val;

$\mathcal{S}_s$: $now[{+}{+}index] \leftarrow rval$; **return** $rval$;

$\mathcal{S}_w$: $now[{+}{+}index] \leftarrow lval$ **or** $rval$; **return** $lval$ **or** $rval$;

$[,)_s$: $now[{+}{+}index] \leftarrow$ (**not** $rval$) **and** $lval$;

return (**not** $rval$) **and** $lval$;

$[,)_w$: $now[{+}{+}index] \leftarrow$ (**not** $rval$); **return** (**not** $rval$);

$\uparrow, \downarrow$: $now[{+}{+}index] \leftarrow val$; **return** $false$;

$\boxdot, \Diamond, \odot$: $now[{+}{+}index] \leftarrow val$; **return** val;

}

}

For a given formula f, we define the function *monitor*, that at each iteration, consumes an event in the run, generates the next state from that event, and evaluates the formula for the state generated:

$monitor(Formula\ f, Run\ r = e_1e_2 \dots e_n)\{$

State $state \leftarrow \{\}$; **array** now, pre;

$state \leftarrow update(state, e_1)$;

$val \leftarrow init(f, state, now, 0)$;

if (**not** val) **then output**(`‘property violated’`);

for $i = 2$ **to** n **do**{

$pre \leftarrow now$;

$state \leftarrow update(state, e_i)$;

$val \leftarrow eval(f, state, now, pre, 0)$;

if (**not** val) **then output**(`‘property violated’`);

}

}

In the initialization phase, the *state* variable is created from the event e_1. The *now* array is then calculated by calling the function *init* on the current *state*. After the calculation, the result of *init* is checked for falsity, and an error message is issued if the result is false. Otherwise, the main loop is started. The main loop goes through the run, starting from the second event. At each iteration, *now* is copied to *pre*, the current state is generated by consuming an event from the run, the formula f is evaluated in the current state using the function *eval*, the result of evaluation is tested for falsity and an error message is generated if the result is false.

The time complexity of this algorithm is $\Theta(mn)$, where m is the size of the original formula and n is the length of the run. However, memory required by the algorithm[1] is $2m'$, m' being the number of temporal and monitor operators in the formula.

Let us return to the Example 1. Suppose that one wants to monitor the safety property $(x > 0) \rightarrow [(y = 0), y > z)_s$ on that program. The formula states that "if $(x > 0)$, then $(y = 0)$ has been true in the past, and since then $(y > z)$ was always false."

For the possible execution or the observed run of the program mentioned in Section 3, we have the following sequence of global states,

$$(-1, 0, 0),\ (0, 0, 0),\ (0, 0, 1),\ (0, 1, 1),\ (1, 1, 1)$$

where the tuple $(-1, 0, 0)$ denotes the state in which $x = -1, y = 0, z = 0$. The formula is satisfied in all the states of the sequence and so we say that the property specified by the formula is not violated by the given run. However, another possible run of the same computation is,

$$\{x = -1, y = 0, z = 0\}, \{x = 0\}, \{y = 1\}, \{z = 1\}, \{x = 1\}$$

This run corresponds to the sequence of states

[1]Here we assume that the recursive version is properly converted into an iterative algorithm using cps transform.

$$(-1,0,0),\ (0,0,0),\ (0,1,0),\ (0,1,1),\ (1,1,1)$$

The formula is violated in the last state of this sequence. Note that, $x > 0$ in the 5th state. This means that from 2nd state, in which $y = 0$, up to 5th state $y > z$ must be false. However, $y > z$ in the 3rd state. This violates the formula. A monitoring algorithm such as the one we describe, which considers only the observed run presented would fail to detect the violation. In the next subsection, we propose an algorithm that will detect such a potential bug from the original successful run.

4.3 Checking Safety against All Runs

The algorithm, presented in the previous subsection, can only monitor a single run. As noticed earlier, monitoring one run may not reveal a bug that might be present in other possible runs. Our algorithm removes this limitation by monitoring all the possible runs of a multithreaded computation. The major hurdle in monitoring all possible runs is that the number of possible runs can be exponential in the length of the computation. We avoid this problem in our algorithm by traversing the computation lattice level by level. The events are generated by the algorithm presented in Section 3. The monitoring module consumes these events one by one, and monitors the safety formula on the computation lattice constructed from the events. We now describe the monitoring module in more detail.

The monitoring module maintains a queue of events. Whenever an event arrives from the running multithreaded program, it *enqueues* it in the event queue (Q). The module also maintains a set of global states ($CurrentLevel$), that are present in the current level of the lattice. For each event e in the event queue, it tries to construct a global state from the set of states in the current level and the event e. If the global state is created successfully, it is added to the set of global states ($NextLevel$) for the next level of the lattice. Once a global state in the current level becomes *unnecessary*, it is removed from the set of global states in the current level. When the set of global states in the current level becomes empty, we say that the set of global states for the next level is *complete*. At that time the module checks the safety formula (by calling *monitorAll(NextLevel)*) for the set of states in the next level. If the formula is not violated, it marks the set of global states for the next level as the set of states for the current level, removes unnecessary events from the event queue, and restarts the iteration. The pseudocode for the process is given below:

for each $(e \in Q)\{$

  if $\exists s \in CurrentLevel$ **s.t.** $isNextState(s, e)$ **then**

    $NextLevel \leftarrow addToSet(NextLevel, createState(s, e));$

  if $isUnnecessary(s)$ **then** $remove(s, CurrentLevel);$

  if $isEmpty(CurrentLevel)$ **then**$\{$

    $monitorAll(NextLevel);$

    $CurrentLevel \leftarrow NextLevel;\ NextLevel \leftarrow \{\};$

    $Q \leftarrow removeUnnecessaryEvents(CurrentLevel, Q);$

  $\}$

$\}$

Every global state s contains the value of all relevant shared variables in the program, a n-dimensional vector clock $VC(s)$ to represent the latest events from each thread that resulted in that global state, and a vector of boolean values called *flags*. Each component of *flags* is initially set to *false*. The predicate *isNextState*(s, e), checks if the event e can convert the state s to a state s' in the next level of the lattice. The pseudo-code for the predicate is given below:

boolean $isNextState(s, e)\{$

  $i \leftarrow threadId(e);$

  if $VC(s)[i] + 1 = VC(e)[i]$ **then**$\{$

    $flags(s)[i] = true;$

    if $(\forall\, 1 \leq j \leq n, j \neq i)\; VC(s)[j] \geq VC(e)[j]$ **then**

return $true$; **else return** $false$; }

else return $false$;

}

where n is the number of threads, $threadId(e)$ returns the index of the thread that generated the event e, $VC(s)$ returns the vector clock of global state s, $VC(e)$ returns the vector clock of event e, and $flags(s)$ returns the vector *flags* associated with s. Note, here $flags(s)[i]$ is set to *true* if $VC(s)[i] + 1 = VC(e)[i]$. This means that e is the only event from thread i that *can possibly* take state s to a state s' in the next level. When all the components of the vector *flags(s)* become *true*, we say that the state s is *unnecessary*. Thus the function $isUnnecessary(s)$ checks if $(\forall 1 \leq i \leq n)$ $flags(s)[i] = true$, where n is the number of threads.

The function $createState(s, e)$ creates a new global state s', where s' is a possible global state that can result from s after the event e. For the purpose of monitoring we maintain, with every global state, a set of *pre* arrays called *PreSet*, and a set of *now* arrays called *NowSet*. In the function *createState* we set the *PreSet* of s' with the *NowSet* of s. The pseudocode for *createState* is as follows:

State $createState(s, e)${

$s' \leftarrow$ **new** $copy\ of\ s$;

$j \leftarrow threadId(e);\ VC(s')[j] \leftarrow VC(s)[j] + 1;$

for $i = 1$ **to** n $\{flags(s')[i] \leftarrow false;\}$

$state(s')[var(e) \leftarrow value(e)];$

$PreSet(s') \leftarrow NowSet(s);$ **return** s';

}

Here $state(s')$ returns the value of all relevant shared variables in state s', $var(e)$ returns the name of the relevant variable that is written at the time of event e, $value(e)$ is the value that is written to $var(e)$, and $state(s')[var(e) \leftarrow value(e)]$ means that in $state(s')$, *var(e)* is updated with *value(e)*.

The function $addToSet(NextLevel, s)$ adds the global state s to the set *NextLevel*. If s

is already present in *NextLevel*, it updates the existing states' *PreSet* with the union of the existing states' *PreSet* and the *PreSet* of s. Two global states are same if their vector clocks are equal. The function $removeUnnecessaryEvents(CurrentLevel, Q)$ removes from Q the events that cannot contribute to the construction of any state at the next level. To do so, it creates a vector clock V_{min} whose each component is the minimum of the corresponding component of the vector clocks of all the global states in the set *CurrentLevel*. It then removes all the events in Q whose vector clocks are less than or equal to V_{min}. This function makes sure that we do not store the unnecessary events.

The function *monitorAll* performs the actual monitoring of the formula. In this function, for each state s in the set *NextLevel*, we invoke the function *eval* (as discussed in the previous section) on s, for each *pre* array in the set *PreSet*. If *eval* returns *false*, we issue a `'property violation'` warning. The *now* array that resulted from the invocation of *eval* is added to the set *NowSet* of s. The pseudocode for the function *monitorAll* is given as follows:

$monitorAll(NextLevel)\{$

  for each $s \in NextLevel\{$

    for each $pre \in PreSet(s)\{$

      $now \leftarrow \{\};\ result \leftarrow eval(f, s, now, pre, 0);$

      if not *result* **then** *output*(`'property violated'`);

      $NowSet(s) \leftarrow NowSet(s) \cup \{now\};\}$

  }

}

If the multithreaded program has synchronization blocks, then we introduce, during instrumentation, a dummy shared variable that is read whenever we enter the synchronization block and is written when we exit the block. This makes sure that all the events in one execution of the block are causally dependent on the events in another execution of the same block, so that the interleaving between them becomes impossible.

Here the size of each *pre* array or *now* array is m', where m' is the number of temporal operators in the formula. Therefore, the size of the set *PreSet* or the set *NowSet* can be at most $2^{m'}$. This implies that the memory required for each state in the lattice is $O(2^{m'})$. If the maximum width of the lattice is w, then the total memory required by the program is $O(w2^{m'})$. The time taken by the algorithm at each iteration is $O(w2^{m'})$. However, if the threads in a program have very few dependency or synchronization points, then the number of valid permutations of the events can be very large, and therefore the width of the lattice can become large. To handle those situations we add a parameter to the algorithm which specifies the maximum width of the lattice: if the number of states in a level of the lattice becomes larger than the maximum width, the algorithm is modified to consider only the most *probable states* in the level.

Chapter 5

Implementation

We have implemented our monitoring algorithm, in a tool called Java Multi PathExplorer (JMPaX)[1], which has been designed to monitor multithreaded Java programs. The current implementation, see Figure 1.1, is written in Java and it assumes that all the shared variables of the multithreaded program are static variables of type `int`. The implementation has around 2000 lines of Java code. It has two main modules, the *instrumentation* module and the *monitoring* module, consisting of around 20 classes. The instrumentation program, named `instrument`, takes a specification file, a port number, and a list of class files as command line arguments. An example of such command is

```
java instrument spec server 7777 A.class B.class C.class
```

where the specification file `spec` contains a list of named formulae. The specification for the example discussed in Section 3 looks as follows:

`F = (A.x > 0) -> [(A.x = 0),(A.y > A.z))`$_s$

where `A` is the class containing the shared variables `x`, `y` and `z` as static fields. The program `instrument` instruments the classes, provided in the argument, as follows:

i) It adds *access* and *write* vector clocks as static fields for each shared variable;

ii) It adds code to create a vector clock whenever a thread is created;

iii) For each read and write access of the shared variables in the class files, it adds codes to update the vector clocks according to the algorithm mentioned in Section 3;

iv) It adds codes to send messages to the `server` at the port number 7777 for all writes of relevant variables. To do so, the `instrument` program extracts the relevant variables from the specification file.

The instrumentation module uses BCEL [7] Java library to modify Java class files. It enables us to insert vector clocks as static member fields in a class, that is otherwise not possible with the tool JTrek [15]. We also make the update of vector clocks associated with a read or write, atomic through `synchronization`. For this we need to add Java bytecode both before and after the instructions `getstatic` and `putstatic`, that access the shared variables. This task is easier in BCEL as compared to JTrek.

A *translator*, which is part of monitoring module, reads the specification and generates a single Java class file, named `SpecificationImpl.class`. The monitoring module starts a server to listen events from the instrumented program, parses them, enqueues them in a queue, executes `translator` to generate `SpecificationImpl.class`, dynamically loads the class `SpecificationImpl.class`, and starts monitoring the formulae on the queue of events. It issues a warning whenever a formula is violated.

Example 1: For the EXAMPLE 1 described in Section 3.5 the safety property of our interest is:

$$(x > 0) \rightarrow [(y = 0), y > z)_s$$

The `SpecificationImpl.java` file generated by the translator for the above property is as follows:

```
package osl.threadtester;

public class SpecificationImpl extends Specification {
    public Formula createFormula() {
        return Implies(AP(1),IntervalS(AP(2),AP(3)));
    }

    public boolean proposition(GlobalStateAndFormula s, int number) {
```

```
        switch(number){
            case 1:
                return s.getValue("test.GlobalVars.x") > 0;
            case 2:
                return s.getValue("test.GlobalVars.y") == 0;
            case 3:
                return s.getValue("test.GlobalVars.y")
                            > s.getValue("test.GlobalVars.z");
        }
        return false;
    }
}
```

Example 2: One of the test cases that we have implemented is the landing example described in Chapter 1. JMPaX was able to detect violation of a safety property from a single execution of the program. The safety property that we verified is:

$$\uparrow\mathtt{landing} \rightarrow [\mathtt{approved}, \downarrow\mathtt{radio})_s.$$

The translator generated the following `SpecificationImpl.java` file.

```
package osl.threadtester;

public class SpecificationImpl extends Specification {

    public Formula createFormula() {
        return Start(AP(1),IntervalS(AP(2),AP(3)));
    }

    public boolean proposition(GlobalStateAndFormula s, int number) {
        switch(number){
            case 1:
                return s.getValue("test.GlobalVars.landing") == 1;
            case 2:
                return s.getValue("test.GlobalVars.approved") == 1;
            case 3:
                return s.getValue("test.GlobalVars.radio") == 1;
        }
        return false;
    }
```

```
}
```

The monitor dynamically loads the compiled class for the above Java code and starts listening for events at port 7777. The *instrument* module properly instrumented the original Java code for the landing controller so that it can send event to the port 7777 while executed. From a single execution of the code in which the radio went off after the landing, the monitoring module constructed a possible run in which radio goes off between landing and approval, and hence it detected the safety violation. This example shows the power of our runtime verification technique.

Chapter 6

Conclusion and Future Work

We have investigated the problem of runtime analysis of multithreaded systems from a fundamental perspective. We have developed scalable techniques for extracting relevant events and their causal dependencies from an executing multithreaded program. We have proposed and implemented algorithms to check safety properties against the computation lattice of a multithreaded computation. We have also briefly presented our prototype tool Java MultiPathExplorer, abbreviated JMPaX, which, at our knowledge, is the first tool that can predict violations of safety properties expressed in temporal logics from correct executions of multithreaded programs. We have also shown that, despite the fact that our safety properties can refer to any state in the past and that there is a potentially exponential number of multithreaded runs to be analyzed, one does not need to actually store the previous states; one can analyze all the multithreaded runs in parallel, by traversing the computation lattice top-down, level-by-level.

In future, we want to extend our technique to analyzing arbitrary distributed systems at runtime. Currently we do not consider distributed systems with message passing because of the fact that past time temporal logic is not a very suitable logic to express properties of distributed systems. Alur et al. [3] show that temporal logic of causality can be good candidate for expressing interesting properties of distributed systems. Meenakshi et al. [20] propose another local logic for message passing systems that we intend to consider in future work. We also intend to investigate the application of the technique to *open distributed* *systems* based on actors [2]. In systems based on actor models, we want to guide the execution of programs so that we can detect a bug quickly.

An extension to this technique is to associate probabilities to the transition system and generate sample runs. The sample runs are then monitored for safety violations using our technique. This gives statistical confidence in our analysis. We are currently investigating different aspects of this statistical extension.

On the monitoring side we want to investigate and design efficient monitors for other different logics that are used most commonly. They include: linear temporal logic for finite traces, extended regular expression, and timed temporal logic. Moreover, there are plans to develop a predictive analysis runtime environment for both multithreaded and distributed systems, as well as to develop a GUI for JMPaX. This would help ordinary software engineers to easily understand and readily use JMPaX. Finally, we plan to use JMPaX on real-world NASA-related large applications.

References

[1] Java Multi PathEXplorer, March 2003. `http://fsl.cs.uiuc.edu/jmpax/`.

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[3] R. Alur, D. Peled, and W. Penczek. Model checking of causality properties. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 90–100, San Diego, California, 1995.

[4] H. W. Cain and M. H. Lipasti. Verifying sequential consistency using vector clocks. In *Proceedings of the 14th annual ACM symposium on Parallel algorithms and Architectures*, pages 153–154. ACM, 2002.

[5] C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[6] 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.

[7] M. Dahm. Byte code engineering with the bcel api. Technical Report B-17-98, Freie Universit at Berlin, Institut für Informatik, April 2001.

[8] J. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

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

[10] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Apr. 2000.

[11] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

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

[13] K. Havelund and G. Roşu. *Runtime Verification 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001. Proceedings of a *Computer Aided Verification (CAV'01)* satellite workshop.

[14] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Proceedings Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, pages 342–356, 2002.

[15] JTrek Compaq Corp. `www.digital.com/java/download/jtrek/`.

[16] 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 *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[17] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based

on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[18] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer-Verlag N.Y., Inc., 1995.

[19] F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier science, 1989.

[20] B. Meenakshi and R. Ramanujam. Reasoning about message passing infinite state environments. In *Proceedings ICALP 00*, volume 1853 of *Lecture Notes in Computer Science*. Springer, 2000.

[21] 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.