Reachability Testing of Concurrent Programs

Yu Lei, *Member*, *IEEE*, and Richard H. Carver

Abstract—One approach to testing concurrent programs, called reachability testing, generates synchronization sequences automatically and on-the-fly, without constructing any static models. In this paper, we present a general execution model for concurrent programs that allows reachability testing to be applied to several commonly used synchronization constructs. We also present a new method for performing reachability testing. This new method guarantees that every partially ordered synchronization sequence will be exercised exactly once without having to save any sequences that have already been exercised. We describe a prototype reachability testing tool called RichTest and report some empirical results, including a comparison between RichTest and a partial order reduction-based tool called VeriSoft. RichTest performed significantly better for the programs in our study.

Index Terms—Software testing, reachability testing, concurrent programming.

1 Introduction

A concurrent program contains two or more threads that execute concurrently and work together to perform some task. Using multiple threads, a.k.a. multithreading, can increase computational efficiency. For instance, while one thread is waiting for user input, another thread can perform computational tasks in the background. In addition, many problem domains are, by nature, concurrent and can be solved more naturally by creating multiple threads. As an example, a Web server typically creates separate threads to service incoming client requests. Some languages, such as Java and Ada, provide built-in support for concurrent programming. The POSIX Pthreads library can be used to write concurrent programs in other languages such as C and C++.

While concurrent programs offer some advantages, they also exhibit nondeterministic behavior, making them notoriously difficult to test. Multiple executions of a concurrent program with the same input may exercise *different* sequences of synchronization events (or SYN-sequences) and may produce *different* results. (The types of synchronization events that can appear in a SYN-sequence include send and receive events on communication channels, P and V events on semaphores, etc. A formal definition of a SYN-sequence is given in Section 3.) One way to deal with nondeterministic behavior during testing is to execute the program with the same input many times and hope that faults will be exposed by at least one of the executions. This type of uncontrolled testing, called *nondeterministic testing*, is easy to carry out, but it can be very inefficient. It is possible that some SYN-sequences of the program are exercised many times whereas others are never exercised at all. *Deterministic testing* is an alternative approach in which executions are controlled so that user-specified SYN-sequences can be exercised. This approach allows a program to be tested with carefully selected SYN-sequences. However, selecting SYN-sequences for deterministic testing is a difficult problem. One commonly suggested approach is to select SYN-sequences from a static model of the program (or its design). However, static models are often inaccurate or may be too large to build.

Reachability testing is an approach that combines nondeterministic and deterministic testing [15]. It is based on a technique called prefix-based testing which executes a test run deterministically up to a certain point and thereafter allows the test run to proceed nondeterministically. The novelty of reachability testing is two-fold. First, reachability testing adopts a dynamic framework in which SYN-sequences are derived automatically and on-the-fly, without constructing any static models. Second, reachability testing is an interleaving-free approach in which independent events are never totally ordered in a SYN-sequence. Therefore, reachability testing automatically avoids the problem of exercising more than one interleaving of the same partial ordering of events. This is in contrast to partial-order reduction techniques [5], [13], [14], which are based on an interleaving-based concurrency model (i.e., a state graph) and try to determine where interleavings can safely be suppressed. In practice, these techniques cannot always avoid exercising more than one interleaving of the same partial ordering of events.

We use an example to illustrate the reachability testing process. Fig. 1a shows a concurrent program CP that consists of four threads. The threads interact by sending messages to, and receiving messages from, ports. Each send operation specifies a port as its destination, and each receive operation specifies a port as its source. Fig. 1b shows an application of reachability testing to CP. It begins by executing CP nondeterministically, which we assume exercises SYN-sequence Q_0. We represent Q_0 as a space-time diagram in which a vertical line represents a thread and a single-headed arrow represents a message passed asynchronously from a send event to a receive event. The

- *Y. Lei is with the Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019. E-mail: ylei@cse.uta.edu.*
- *R.H. Carver is with the Department of Computer Science, George Mason University, Fairfax, VA 22030. E-mail: rcarver@cs.gmu.edu.*

Manuscript received 15 Apr. 2005; revised 13 Nov. 2005; accepted 29 Mar. 2006; published online 23 June 2006.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0092-0405.

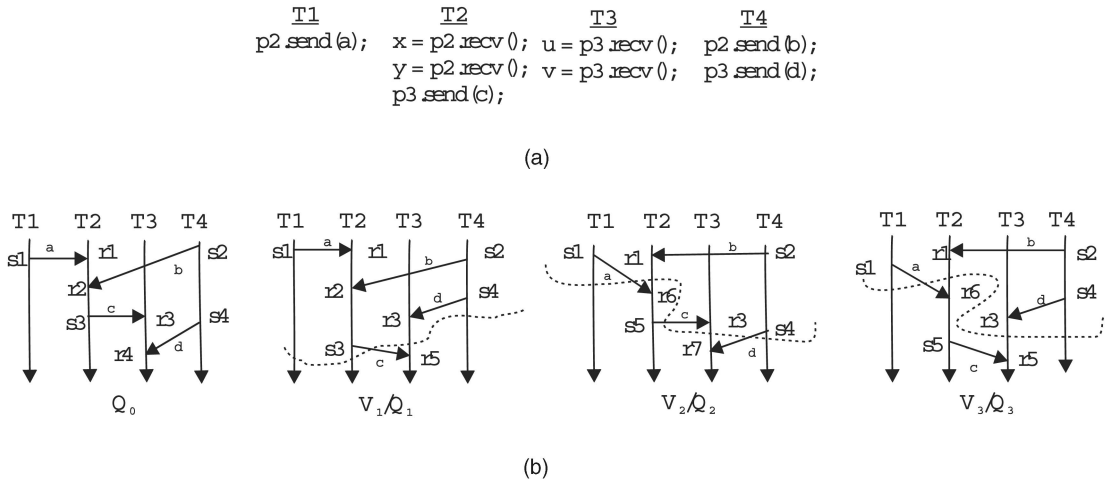

Fig. 1. An example program and a reachability testing scenario of the program.

race conditions in Q_0 are then identified and used to derive "race variants," namely, V_1, V_2, and V_3, of Q_0. Each race variant is a prefix of Q_0 with the outcome of one or more race conditions changed. (We will explain how to derive these race variants in Section 6.) Each variant is used to conduct a prefix-based test run, which forces the events and synchronizations in the variant to be replayed and then allows the test run to proceed nondeterministically. Prefix-based testing with V_1, V_2, and V_3 exercises complete sequences Q_1, Q_2, and Q_3, respectively. (In Fig. 1b, the sequence Q_i and the variant V_i used to exercise it are shown in the same space-time diagram. The events and synchronizations in the variant are above the dashed line. The naming of the events will be explained in Section 5.) No new variants can be derived from Q_1, Q_2, and Q_3, so the reachability testing process stops. Note that Q_0, Q_1, Q_2, and Q_3 are all the (partially ordered) SYN-sequences the example program can possibly exercise.

In this paper, we assume that the program under test is closed, i.e., the environment that the program interacts with is modeled as part of the program, and the only source of nondeterminism is due to concurrency, i.e., the order in which threads synchronize and communicate. We also assume that a test driver is used to make every test run terminate. Under these assumptions, reachability testing of a concurrent program CP with a given input X will exercise every partially ordered SYN-sequence of CP with input X. Note that data inputs can be selected using techniques such as domain partitioning, which is not discussed in this paper. It is worth noting that, for some programs, e.g., programs that implement certain network protocols, the synchronization behavior is usually independent of data inputs.

The main contributions of this paper are as follows: First, we present a general execution model for concurrent programs that allows reachability testing to be applied to several commonly used synchronization constructs, including asynchronous and synchronous message passing, semaphores, and monitors. Second, we show how to extend traditional schemes for timestamping events in a way such that race conditions can be identified in an execution that is modeled using our general execution model. Third, we present a new algorithm for computing the race variants of a SYN-sequence. Existing algorithms for computing race variants are interleaving-based, i.e., they generate race variants that represent all the possible interleavings of events and then filter out those that correspond to the same partial order [15], [29]. Our algorithm generates partially ordered race variants in which independent events are never totally ordered and is thus more efficient than existing algorithms. Fourth, we present a new reachability testing algorithm for driving the testing process. In order to avoid exercising the same SYN-sequence more than once, all existing reachability testing algorithms need to save the history of SYN-sequences that have been exercised. Our new algorithm saves no SYN-sequences, but still guarantees that every partially ordered SYN-sequence is exercised exactly once. This significantly reduces the space and time requirements of reachability testing. Finally, we describe a prototype reachability testing tool, called RichTest, and report some empirical results. RichTest is written in Java and makes no modification to the Java virtual machine or to the underlying operating system.

The rest of this paper is organized as follows: Section 2 gives an overview of several commonly used synchronization constructs. Section 3 presents a general execution model for these synchronization constructs. Section 4 shows how to assign timestamps to events in our execution model. Section 5 shows how to identify the race conditions in an execution. Section 6 describes our new algorithm for computing race variants. Section 7 presents our new reachability testing algorithm for driving the testing process. Section 8 describes the RichTest tool and reports some empirical results. Section 9 briefly surveys related work. Section 10 provides concluding remarks and describes our plans for future work.

2 Overview of Synchronization Constructs

In this section, we briefly describe several commonly used synchronization constructs, including asynchronous and synchronous message passing, semaphores, and monitors. We assume that the queues associated with these constructs are First-In-First-Out (FIFO), but the reachability testing

algorithms presented later do not depend on this assumption; other queuing disciplines can be used.

2.1 Asynchronous Message Passing

Asynchronous message passing refers to nonblocking send operations and blocking receive operations. A thread that executes a nonblocking send operation proceeds without waiting for the message to arrive at its destination. A thread that executes a blocking receive operation blocks until a message is received. As shown below, each send operation specifies a destination port to which its message will be sent and each receive operation specifies a source port from which a message will be retrieved. Note that a port is a communication object that has multiple senders but only one receiver.

```
Port p;
Thread1          Thread2
p.send(msg)      msg = p.receive();
```

We assume that ports use a FIFO (First-In-First-Out) message passing scheme, which guarantees that two messages sent by the same thread to a given port are received from that port in the order in which they are sent. In practice, ports are often implemented using bounded buffers that can only hold a limited number of messages. In this case, a send operation can be blocked if the capacity of the buffer is reached.

2.2 Synchronous Message Passing

Synchronous message passing is the term used when send and receive operations are both blocking. The receiving thread blocks until a message is received. The sending thread blocks until it receives an acknowledgment that the message it sent was received by the receiving thread.

Selective wait statements are commonly used with synchronous message passing to allow a combination of waiting for, and selecting from, one or more $receive()$ alternatives [1]. The selection depends on guard conditions associated with each alternative of the selective wait:

```
Port p1, p2;
select
   when (guard condition 1) => p1.receive();
or
   when (guard condition 2) => p2.receive();
end select;
```

A receive-alternative is said to be *open*, and thus selectable, if it does not have a guard condition or if the value of the guard condition is *true*. Otherwise, the alternative is said to be *closed* and it cannot be selected. We restrict selective waits to having at most one receive-alternative for a given port. Also, we assume that the choice among multiple open alternatives in a selective wait is based on the order in which messages arrive. (Messages arrive at their destination some time after they are sent and they are queued until they are received.)

2.3 Semaphores

A semaphore is a synchronization object that is initialized with an integer value and is accessed through two operations named P and V. For a *counting semaphore* s, at any time, the following relation, called the *semaphore invariant*, holds:

$$\begin{aligned}&(initial\ value\ of\ s)\\ &\quad + (number\ of\ completed\ s.V()\ operations)\\ &\geq (number\ of\ completed\ s.P()\ operations).\end{aligned}$$

A thread that starts a $P()$ operation may be blocked inside $P()$, so the operation may not be completed right away. The invariant refers to the number of *completed* operations, which may be less than the number of *started* operations. For a counting semaphore, $V()$ operations never block their caller and are always completed immediately.

For a *binary semaphore* initialized to 1 (0), the first completed operation must be a $P()$ ($V()$) operation and the completion of $P()$ and $V()$ operations must alternate after that. Thus, the $P()$ and $V()$ operations of a binary semaphore may block the calling threads [2]. We assume that the queues of blocked threads associated with counting and binary semaphores are FIFO queues.

2.4 Monitors

A monitor is a high-level synchronization construct that supports data encapsulation and information hiding. At most one thread is allowed to execute inside a monitor at any time. Mutual exclusion is enforced by the monitor's implementation, which ensures that each monitor method is a critical section. Conditional synchronization is achieved using condition variables and operations $wait()$ and $signal()$. A condition variable denotes a queue of threads that are waiting to be signaled that a specific condition is true. (The condition is not explicitly specified as part of the condition variable.) There are several different types of signaling disciplines [2]. When the *Signal-and-Continue* (*SC*) discipline is used, the signaling thread continues to execute in the monitor and the signaled thread has to compete with other threads to reenter the monitor. When the *Signal-and-Urgent-Wait* (*SU*) discipline is used, the signaling thread exits the monitor and the signaled thread reenters the monitor immediately. We assume that the queues associated with a monitor and its condition variables are FIFO queues.

3 General Execution Model for Reachability Testing

In this section, we present a general execution model for the synchronization constructs described in Section 2. This model provides sufficient information for replaying an execution and for identifying race conditions in an execution. Replay techniques have already been developed for these constructs [7], [30]. Our execution model contains all the information required by these techniques. In Section 5, we will discuss how to identify the race conditions in an execution that is represented using our model.

We first show, for each synchronization construct, what events are modeled in a concurrent execution:

- *Asynchronous and synchronous message passing*. When a thread T performs a send (receive) operation, a send (receive) event occurs on T.

TABLE 1
Event Descriptors for a Sending Event s

Synchronization construct	Sender	Destination	Operation	i
asynchronous message passing	sending thread ID	Port ID	send	event index
synchronous message passing	sending thread ID	Port ID	send	event index
semaphores	calling thread ID	semaphore ID	P or V	event index
monitors	calling thread ID	monitor ID	method name	event index

TABLE 2
Event Descriptors for a Receiving Event r

Synchronization construct	Receiver	OpenList	i
asynchronous message passing	receiving thread ID	the port of r	event index
synchronous message passing	receiving thread ID	open ports (including r's port)	event index
semaphores	semaphore ID	operations (P and/or V) that could be completed at r	event index
monitors	monitor ID	all of the monitor's methods	event index

- *Semaphore*. When a thread T calls a $P()$ or $V()$ operation on a semaphore s, a semaphore *call* event occurs on T. When a $P()$ or $V()$ operation on a semaphore s is completed, a semaphore *completion* event occurs on s.
- *SU monitor*. When a thread T calls a method of monitor M, a monitor *call* event occurs on T. When T eventually enters M, a monitor *entry* event occurs on M and then T starts to execute inside M. Note that reentries into an SU monitor are not modeled since they do not compete with other threads and, thus, do not involve any race conditions.

  SC monitor. When a thread T calls a method of monitor M, a monitor *call* event occurs on T. A monitor *call* event also occurs when T tries to reenter a monitor M after being signaled. When thread T eventually (re)enters M, a monitor *entry* event occurs on M and T starts to execute inside M.

In our general execution model, we refer to a *send* or *call* event as a *sending event*, and a *receive, completion*, or *entry* event as a *receiving event*. We refer to a semaphore or monitor generally as a synchronization object. If a sending event s is synchronized with a receiving event r in an execution (i.e., a sent message is received or a called semaphore operation is completed or a called monitor method is entered), we refer to $< s, r >$ as a *synchronization pair* and say that s is the sending partner of r and r is the receiving partner of s.

Definition 1. *The SYN-sequence Q exercised by a concurrent execution is defined as a tuple $(Q_1, Q_2, \ldots, Q_n, \phi)$, where Q_i is the totally ordered sequence of sending and receiving events that occurred on a thread or a synchronization object and ϕ is the set of synchronization pairs exercised in the execution.*

The notion of a SYN-sequence has been defined for many different concurrent programming languages and constructs [9] and has been used for the specification and testing of concurrent programs [7], [8], [30], [31]. Note that the outcome of a concurrent execution is determined by the program text, the input, and the SYN-sequence exercised by the execution. In this paper, we deal with nondeterministic executions of the *same* program with the *same* input. Thus, we will characterize a concurrent execution simply as the SYN-sequence exercised by the execution. Unless otherwise specified, all the SYN-sequences are assumed to be exercised by executions of the same program with the same input.

Definition 2. *Two SYN-sequences Q and Q' are equal, denoted as $Q = Q'$, if there exists a one-to-one mapping m from the events in Q to those in Q' that preserves the synchronization relation, i.e., $< m(s), m(r) >$ is a synchronization pair in Q' if and only if $< s, r >$ is a synchronization pair in Q.*

As shown in Fig. 1, a SYN-sequence is often depicted using a space-time diagram. Intuitively, two sequences are equal if their space-time diagrams are the same except for possibly their event labels. We will use $send(r, Q)$ to denote the sending partner of a receiving event r, if the sending partner exists, in a SYN-sequence Q. Note that $send(r, Q)$ is undefined if r is not synchronized with any sending event in Q. To encode certain information about each event, we introduce the notion of an event descriptor:

- A descriptor for a sending event s is denoted by $(Sender, Destination, Operation, i)$, where $Sender$ is the thread executing the sending event, $Destination$ is the destination thread or synchronization object, $Operation$ is the type of the operation performed (P, V, $send$, $receive$, etc.), and i is the event index indicating that s is the ith event exercised by the sending thread.
- A descriptor for a receiving event r is denoted by $(Receiver, OpenList, i)$, where $Receiver$ is the receiving thread or object, $OpenList$ is a field to be defined later that assists in identifying race conditions, and i is the event index indicating that r is the ith event occurring on the receiving thread or object.

The individual fields of an event descriptor are referenced using dot notation. For example, the $Operation$ of a sending event s is referred to as $s.Operation$.

Table 1 and Table 2 summarize the specific information that is contained in the event descriptors for the

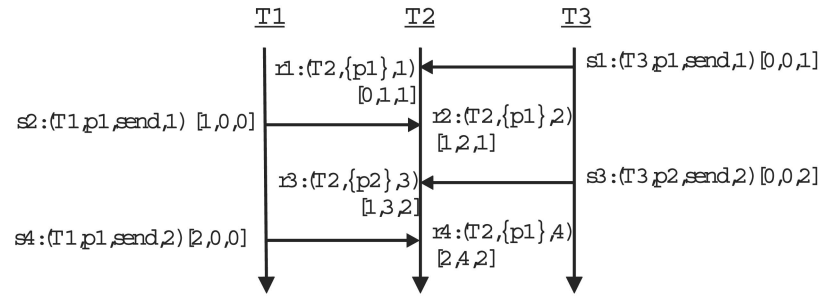

Fig. 2. An asynchronous message passing execution.

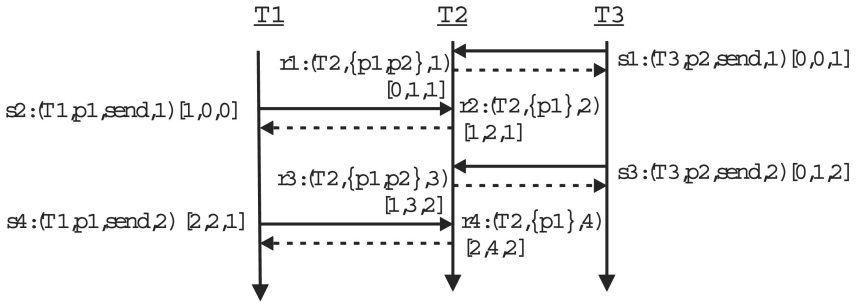

Fig. 3. A synchronous message passing execution.

synchronization constructs described in Section 2. Although the information is construct-specific, the format, as well as the general meaning of each field in the event descriptor, is the same for all constructs. This allows us to present a single race analysis algorithm that operates on event descriptors and thus works for all the constructs. The values for the fields in the event descriptors are straightforward, except for the $OpenLists$ of the receiving events, which are discussed below.

Fig. 2, Fig. 3, Fig. 4, and Fig. 5 show an example execution for each of the synchronization constructs. In each diagram, the event descriptor of an event is shown beside the name of the event and inside a pair of parentheses. A solid arrow is drawn from a sending event s to a receiving event r, indicating that $< s, r >$ is a synchronization pair. The brackets as well as the dashed arrows will be explained in Section 4. Below we use Fig. 2, Fig. 3, Fig. 4, and Fig. 5 to illustrate how to compute the $OpenList$ of a receiving event.

- *Asynchronous message passing*. The $OpenList$ of a receive event r contains a single port, which is the source port of r. A send event s is said to be open at r if port $s.Destination$ is in the $OpenList$ of r, which means that the ports of s and r match. Fig. 2 shows an asynchronous message passing execution. Thread $T1$ sends two messages to port $p1$. Thread $T3$ sends its first message to port $p1$ and its second message to port $p2$. Thread $T2$ first receives two messages from port $p1$, followed by one message from port $p2$, and then another message from port $p1$. In Fig. 2, $s2$ is open at $r1$ since the ports of $s2$ and $r1$ match.
- *Synchronous message passing*. The $OpenList$ of a receive event r that occurs inside a selective wait is the list of ports that had open receive-alternatives when r was selected. Note that this list always includes the source port of r. For a receive event r that does not occur inside a selective wait, the $OpenList$ contains only the source port of r. Event s is said to be *open* at r if the destination port of s is in the $OpenList$ of r. Fig. 3 shows a synchronous message passing execution. Thread $T1$ sends two messages to port $p1$ and thread $T3$ sends two messages to port $p2$. Thread $T2$ executes a selective wait with receive-alternatives for $p1$ and $p2$. In Fig. 3, the $OpenLists$ for the receive events indicate that, during execution, each time the receive-alternative for $p2$ was selected, the receive-alternative for $p1$ was open and each time the receive-alternative for $p1$ was selected, the receive-alternative for $p2$ was closed.
- *Semaphores*. For a completion event e that represents the completion of a $P()$ or $V()$ operation, the $OpenList$ of e specifies the types of operations (P and/or V) that could be completed at e. $OpenLists$ can easily be computed at runtime based on the semaphore invariant. A call event c is open at a completion event e if $c.Destination = e.Receiver$ and $c.Operation$ is in $e.OpenList$. Fig. 4 shows a semaphore-based execution with threads $T1$ and

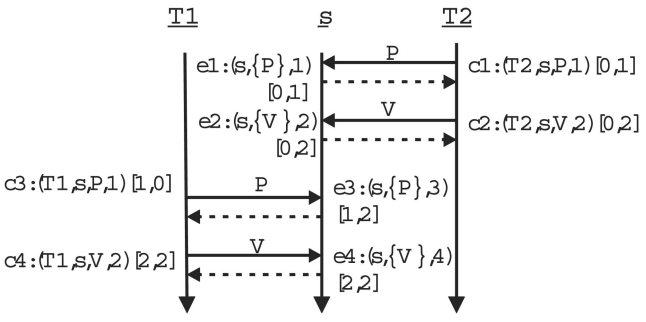

Fig. 4. A semaphore-based execution.

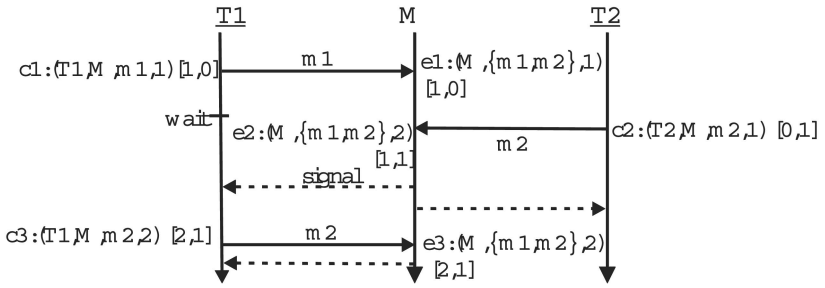

Fig. 5. A monitor-based execution.

$T2$ and a binary semaphore s initialized to 1. $T1$ and $T2$ both perform a $P()$ and a $V()$ operation on s. In this diagram, semaphore s is also represented as a vertical line, which contains the completed P and V events that occur on s. In Fig. 4, the $OpenLists$ for the receiving events model the fact that the completion of P and V operations on a binary semaphore must alternate. That is, the $OpenList$ for an event representing a P(V) operation only contains P(V), not V(P).

- *Monitors*. The $OpenList$ of a monitor is the list of all of the (public) methods defined on the monitor since entry into a monitor is never guarded. A call event c is open at an entry event e if the called monitor of c is the owning monitor of e, i.e., $c.Destination = e.Receiver$. Fig. 5 shows a monitor-based execution involving threads $T1$ and $T2$ and an SC monitor M. Two methods, $m1$ and $m2$, are defined on monitor M. Thread $T1$ enters M first by calling $m1$ and executes a $wait()$ operation. Then, thread $T2$ enters M by calling $m2$, executes a $signal()$ operation that signals $T1$, and then exits M. Note that event $c3$ occurs when $T1$ tries to reenter M after being signaled by $T2$. In Fig. 5, the $OpenList$ of each entry event always includes methods $m1$ and $m2$ defined in monitor M.

Note that we have shown in [6] that our general execution model can also be applied to locks, which are another commonly used synchronization construct. Since locks and binary semaphores can be handled similarly, and due to space constraints, locks are not discussed in this paper. Our general model can also be used to handle shared variables by modeling a read (write) operation as receiving (sending) a message from (to) a shared variable. Again, due to space constraints, details of this extension are not discussed in this paper. If a program correctly protects accesses to shared variables using a semaphore (which can be a binary semaphore or a counting semaphore whose initial value is 1) or a monitor, then the program can be handled by our general model as a semaphore or monitor-based program.

4 Timestamp Assignment

In this section, we show how to assign vector timestamps to the events in an execution that is modeled using our general execution model. These timestamps can be used to determine the *happened-before* relation between events. This relation is needed to identify the race conditions in an execution, as shown in Section 5. Intuitively, an event e_1 *happened before* another event e_2 in a *SYN*-sequence Q if e_1 could potentially affect e_2 [18]. We denote this as $e_1 \rightarrow_Q e_2$ or, simply, $e_1 \rightarrow e_2$ if Q is implied.

A vector timestamp scheme for asynchronous message passing has already been developed [11], [23]. In this scheme, each thread maintains a vector clock. A vector clock is a vector of integers that are used to keep track of the integer clock of each thread. The integer clock of a thread is initially zero and is incremented each time the thread executes a send or receive event. Each send and receive event is also assigned a vector timestamp. In the following, we will use $v[i]$ to denote the ith component of vector v and $max(v_1, v_2)$ to denote the componentwise maximum of vectors v_1 and v_2.

Let $T.v$ be the vector clock maintained by a thread T. Let $e.ts$ be the vector timestamp of an event e. The vector clock of a thread is initially a vector of zeros. The following rules are used to update vector clocks and assign timestamps to the send and receive events for asynchronous message passing:

1. When a thread T_i executes a nonblocking send event s, it performs the following operations: a) $T_i.v[i] = T_i.v[i] + 1$ and b) $s.ts = T_i.v$. The message sent by s also carries the timestamp $s.ts$.
2. When a thread T_j executes a receive event r that receives the message sent by s, it performs the following operations: a) $T_j.v[j] = T_j.v[j] + 1$, b) $T_j.v = max(T_j.v, s.ts)$, and c) $r.ts = T_j.v$.

In Fig. 2, the vector timestamp of each event that is assigned using the above scheme is shown inside a pair of brackets.

A vector timestamp scheme for synchronous message passing has also been developed [11], but this scheme must be extended for race analysis. The traditional timestamp scheme for synchronous message passing assigns the same timestamp to the send and receive events in a synchronization pair:

1. When a thread T_i executes a blocking send event s, it performs the following operation: $T_i.v[i] = T_i.v[i] + 1$. The message sent by s also carries the vector clock $T_i.v$.
2. When a thread T_j executes a receive event r that receives the message sent by s, it performs the following operations: a) $T_j.v[j] = T_j.v[j] + 1$, b) $T_j.v = max(T_j.v, T_i.v)$, and c) $r.ts = T_j.v$. Thread T_j also sends $T_j.v$ back to thread T_i.

3. Thread T_i receives $T_j.v$ and performs the following operations a) $T_i.v = max(T_i.v, T_j.v)$ and b) $s.ts = T_i.v$.

Let s be a send event and r a receive event such that $< s, r >$ is a synchronization pair. In the above scheme, s and r are assigned with the same timestamp and are considered to happen at the same time. However, in order to support race analysis, synchronous send and receive events should not be considered to happen at the same time. Assume that two threads $T1$ and $T2$ each send a message to another thread $T3$ and they send these messages at the same time. Let s and s' be the two send events executed by $T1$ and $T2$, respectively. Suppose that $T3$ receives the message sent by $T1$ first and then the message sent by $T2$. Let r and r' be the two receive events executed by $T3$, in that order, meaning that $< s, r >$ and $< s', r' >$ are two synchronization pairs. In the traditional timestamp scheme, s' will be considered to happen at the same time as r', which happens after r and, thus, also happens after s, whereas the messages were actually sent by $T1$ and $T2$ at the same time and both s and s' could be synchronized with r. Therefore, we need to make a slight modification to the above scheme. Our new scheme for synchronous message passing is shown below.

1. When a thread T_i executes a blocking send event s, it performs the following operations: a) $T_i.v[i] = T_i.v[i] + 1$ and b) $s.ts = T_i.v$. The message sent by s also carries the timestamp $s.ts$.
2. When a thread T_j executes a receive event r that receives the message sent by s, it performs the following operations: a) $T_j.v[j] = T_j.v[j] + 1$, b) $T_j.v = max(T_j.v, s.ts)$, and c) $r.ts = T_j.v$. Thread T_j also sends $T_j.v$ back to thread T_i.
3. Thread T_i receives $T_j.v$ and performs $T_i.v = max(T_i.v, T_j.v)$.

Note that, in the above scheme, the vector clocks of T_i and T_j are exchanged, but the timestamps of s and r are not. In Fig. 3, the timestamp of each event is shown inside a pair of brackets. Note that the dashed arrows represent applications of rule 3.

Next, we describe a timestamp scheme for semaphores and monitors. In this scheme, each thread and synchronization object maintains a vector clock. As before, position i in a vector clock refers to the integer clock of thread T_i. Synchronization objects maintain a vector clock, but they do not have integer clocks; thus, a synchronization object does not have a position in a vector clock. Let $T.v$ (or $O.v$) be the vector clock maintained by thread T (or synchronization object O). The vector clock of a thread or synchronization object is initially a vector of zeros. The following rules are used to update vector clocks and assign timestamps to events:

1. When a thread T_i executes a sending event s, it performs the following operations: a) $T_i.v[i] = T_i.v[i] + 1$ and b) $s.ts = T_i.v$.
2. When a receiving event r that is synchronized with a sending event s occurs on a synchronization object O, the following operations are performed: a) $O.v = max(O.v, s.ts)$ and b) $r.ts = O.v$.
3. Semaphore: When a thread T_i finishes executing a $P()$ or $V()$ operation on semaphore O, it updates its vector clock using the componentwise maximum of $T_i.v$ and $O.v$, i.e., $T_i.v = max(T_i.v, O.v)$.

   SU monitor: When a thread T_i finishes executing a method of monitor O, it updates its vector clock using the componentwise maximum of $T_i.v$ and $O.v$, i.e., $T_i.v = max(T_i.v, O.v)$.

   SC monitor: When a thread T_i finishes executing a method of monitor O or when T_i is signaled while waiting on a condition variable of O, it updates its vector clock using the componentwise maximum of $Ti.v$ and $O.v$, i.e., $T_i.v = max(T_i.v, O.v)$.

In Fig. 4 and Fig. 5, the timestamp of each event is shown inside a pair of brackets. Again, dashed arrows represent applications of rule 3.

Proposition 1 shows how to use vector timestamps to determine the happened-before relation between two arbitrary events. This proposition is an extension of the results in [11], [23] for covering semaphore and monitor-based executions.

Proposition 1. *Let X be an execution involving threads $T_1, T_2, \ldots, T_n$. Let Q be the SYN-sequence exercised by X. Assume that every event in Q is assigned a vector timestamp as described above. For a sending or receiving event e, let $e.tid$ be the (integer) thread ID of the thread that executes e. (If event e is a completion event on a semaphore or an entry event on a monitor, then $e.tid$ is the thread ID of the thread that executes the sending partner of e.) Let e_1 and e_2 be two events in Q. Then, $e_1 \rightarrow e_2$ if and only if 1) $< e_1, e_2 >$ is a synchronization pair or 2) $e_1.ts[e_1.tid] \leq e_2.ts[e_1.tid]$ and $e_1.ts[e_2.tid] < e_2.ts[e_2.tid]$.*

5 Race Analysis

In this section, we show how to identify the race conditions in a concurrent execution. As shown later, a race condition is a phenomenon involving an execution and the possible alternatives of this execution. Thus, we need to draw a correspondence between the events in the original and alternative executions. For this purpose, we introduce the notion of event equality, which formally defines what it means for an event in one execution to be the same as, or equal to, an event in another execution.

5.1 Event Equality

We define event equality based on the "control-structure" of an event. Informally, the control-structure of event e contains all the events, as well as the synchronizations between them, that could possibly *control* whether or not event e is executed. Fig. 6a shows a SYN-sequence Q for an asynchronous message passing execution. Send event s_3 happened before receive event r_3 since (s_3, r_3) is a synchronization pair. (Recall that all notions of time used in this section are relative to the *happened-before* relation defined in Section 4.) Receive event r_1 also happened before r_3 since T2 executes r_1 before it executes r_3. However, there is a subtle distinction between the happened-before relation between s_3 and r_3 and that between r_1 and r_3. Let L be the statement whose execution gives rise to the occurrence of r_3.

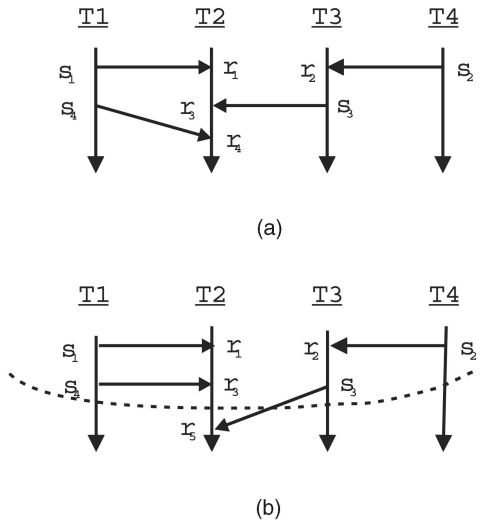

Fig. 6. (a) Sequence Q. (b) Variant V_1 of Q and sequence Q_1.

Then, whether or not statement L is executed, i.e., whether or not r_3 occurs, may depend on the control exerted by the execution of events s_1 and r_1, but it does not depend on the execution of s_3. Therefore, events r_1 and s_1 are said to be in the "control-structure" of r_3, whereas s_3 is not considered to be in the control-structure of r_3.

Definition 3. *Let e be an event exercised by a thread T in a SYN-sequence Q. Then, the control-structure of e in Q, denoted as $c\text{-}struct(e, Q)$, is empty if e is the first event exercised by T; otherwise, it is the prefix of Q that contains the event f that T exercised immediately before e and all the events that happened before f, including the synchronizations between these events.*

Note that the control-structure of a sending event s consists of all the events that happened before s, whereas the control-structure of a receiving event r may not include all the events that happened before r. As an example, in sequence Q in Fig. 6a, $c\text{-}struct(s_3, Q)$ contains r_2 and s_2, whereas $c\text{-}struct(r_3, Q)$ contains s_1 and r_1, but not s_3, s_2, and r_2.

The following definition shows that two events are equal if their control-structures are equal. Note that the control-structure of an event is also a SYN-sequence and the equality of control structures can be determined by Definition 2.

Definition 4. *Let CP be a concurrent program. Let Q and Q' be two SYN-sequences of CP with input X. Let e be an event in Q and e' an event in Q'. Events e and e' are equal, denoted as $e = e'$, if $c\text{-}struct(e, Q) = c\text{-}struct(e', Q')$.*

Fig. 6b shows a race variant V_1 of Q and a SYN-sequence Q_1 collected from prefix-based testing with V_1, where V_1 is the portion of Q_1 that is above the dashed line. Note that equal events in Q and Q_1 are given the same event label. As an example, event s_4 in Q equals event s_4 in Q_1 since both $c\text{-}struct(s_4, Q)$ and $c\text{-}struct(s_4, Q_1)$ contain only one send event, namely, s_1, and thus are equal to each other. As another example, event r_3 in Q equals event r_3 in Q_1 because both $c\text{-}struct(r_3, Q)$ and $c\text{-}struct(r_3, Q_1)$ contain two events, namely, s_1 and r_1, and $< s_1, r_1 >$ is a synchronization pair in both Q and Q_1. We will consider equal events in different SYN-sequences to be the same event and always give equal events the same label. Note that this is how the events in Fig. 1 are labeled. Doing so allows us to refer to the control-structure of an event e without referring to any one of the possibly many SYN-sequences that e may appear in, i.e., we can simply write $c\text{-}struct(e)$.

Note that Definition 4 is a conservative definition of event equality. For example, Q and Q_1 in Fig. 6 are different sequences. Event r_5 in Q_1 is a new event that does not appear in Q. Event r_5 was caused by changing r_3's sending partner from s_3 to s_4 in variant V_1. Note that events r_4 and r_5 might be generated by the same receive statement in Thread2 and r_3 may receive from s_4 the same value that it received from s_3, but r_4 and r_5 are still not considered to be equal events. This reflects the fact that Q and Q_1 are considered to be different SYN-sequences even though the outcome of the program might be the same, regardless of which sequence is exercised. Some discussion on how to avoid exercising equivalent sequences is provided at the end of Section 7.

5.2 Computing Race Sets

Intuitively, there exists a race condition or simply a race between two sending events if they can be synchronized with the same receiving event in different executions. Note that races are caused by factors such as variations in thread scheduling and message delays. In order to accurately determine all the races in an execution, the program's logic must be analyzed. Detecting races using static analysis is undecidable for arbitrary programs [4] and is NP-complete for even very restricted classes of programs (e.g., those containing no branches) [32]. However, for the purpose of reachability testing, we only need to consider a special type of race, called a lead race. Lead races can be identified by analyzing the SYN-sequence of an execution, i.e., without analyzing the program's logic.

Definition 5. *Let Q be the SYN-sequence exercised by an execution of a concurrent program CP with input X. Let s be a sending event and r be a receiving event in Q such that $< s, r >$ is a synchronization pair. Let s' be another sending event in Q. There exists a lead race between s' and s with respect to r in Q if s' and r can be synchronized with each other during another execution of CP with input X in which all the events that happen before s' or r in Q, as well as the synchronizations between these events, are replayed.*

Definition 5 requires all the events that can potentially affect s' or r in Q to be replayed and thus guarantees that s' and r will be exercised in the alternative execution. In the rest of this paper, a race is assumed to be a lead race unless otherwise specified.

Recall that the control-structure of a receiving event r does not include its sending partner. Otherwise, if the control-structure of r was defined to include its sending partner, then, whenever r was synchronized with a different sending partner, r's control-structure would be different and r would become a *different* event. As a result, we would not be able to express the phenomenon that the

same receiving event can be synchronized with *different* sending partners.

Definition 6. *Let Q be a SYN-sequence. Let s be a sending event and r be a receiving event in Q such that $< s, r >$ is a synchronization pair. The race set of r in Q, denoted as $race\_set(r, Q)$ or $race\_set(r)$ if Q is implied, is the set of sending events in Q that have a race with s w.r.t. r. Formally,*

$$race\_set(r, Q) = \{s' \in Q \mid there\ exists\ a\ race\ between\ s'\ and\ s\ with\ respect\ to\ r\}.$$

Proposition 2 describes how to compute the race set of a receiving event. This proposition is an extension of the result in [28] for covering synchronous message passing, semaphore, and monitor-based executions.

Proposition 2. *Let Q be a SYN-sequence. A sending event s is in the race set of a receiving event r in Q if*

1. *s is open at r,*
2. *r does not happen before s,*
3. *if $< s, r' >$ is a synchronization pair, then r happens before r', and*
4. *if a sending event s' has the same source and destination as s but happens before s, then there exists a receiving event r' such that $< s', r' >$ is a synchronization pair and r' happens before r.*

The last condition in the above proposition deserves some explanation. For asynchronous and synchronous message passing, this condition reflects the assumed *FIFO* message ordering scheme. That is, since messages sent from the same thread to the same port are received in the order in which they are sent, the message sent by s cannot be received by r unless the message sent by s' has been received before r. Similarly, for semaphores and monitors, this condition reflects the fact that calls to $P()$ and $V()$ operations and monitor methods that are made by the same thread are completed in the order in which they are called.

Below, we show the race set of each receiving event in the example executions in Fig. 2, Fig. 3, Fig. 4, and Fig. 5:

- *Asynchronous message passing*. The race set of each receive event in Fig. 2 is as follows:

  $$race\_set(r1) = \{s2\},$$
  $$race\_set(r2) = race\_set(r3) = race\_set(r4) = \{\}.$$

  Note that $s3$ is not in $race\_set(r2)$ because $s3$ is sent to a different port and, thus, $s3$ is not open at $r2$. For the same reason, $s4$ is not in $race\_set(r3)$. Also note that $s4$ is not in $race\_set(r1)$ because the FIFO message ordering scheme requires $s2$ to be synchronized with $r1$ before $s4$.
- *Synchronous message passing*. The race set of each receive event in Fig. 3 is as follows:

  $$race\_set(r1) = \{s2\}, race\_set(r2) = \{\},$$
  $$race\_set(r3) = \{s4\}, \text{and } race\_set(r4) = \{\}.$$

  Note that $s3$ is not in $race\_set(r2)$ because $p2$ is not in the $OpenList$ of $r2$ and, thus, $s3$ is not open at $r2$.
- *Semaphores*. The race set of each completion event in Fig. 4 is as follows:

  $$race\_set(e1) = \{c3\} \text{ and}$$
  $$race\_set(e2) = race\_set(e3) = race\_set(e4) = \{\}.$$

  Note that, since P was not in the $OpenList$ of $e2$, $c3$ is not in $race\_set(e2)$. This captures the fact that the $P()$ operation by $T1$ could start but not complete before the $V()$ operation by $T2$ and, hence, that these operations do not race.
- *Monitors*. The race set of each entry event in Fig. 5 is as follows:

  $$race\_set(e1) = \{c2\},$$
  $$race\_set(e2) = race\_set(e3) = \{\}.$$

  Sending event $c3$ is not in $race\_set(e2)$ since $c3$ happened after $e2$. This captures the fact that $T2$ entered monitor M at $e2$ and executed a signal operation that caused $T1$ to issue $c3$ and, thus, there is no race between $c2$ and $c3$ with respect to $e2$.

6 Computing Race Variants

In this section, we first define the notion of a race variant and then we present an algorithm for computing the race variants of a SYN-sequence.

6.1 Race Variant

Let CP be a concurrent program. Let Q be the SYN-sequence exercised by an execution of CP with input X. Informally, a race variant of Q is derived by changing the sending partner of one or more receiving events in Q in a way that satisfies the following constraints: 1) If we change the sending partner of a receiving event r, the new sending partner must be an event in the race set of r. 2) If and only if we change the sending partner of a receiving event r do we remove all the events whose control structures contain r.

Definition 7. *Let Q be a SYN-sequence. A race variant V of Q is another SYN-sequence that satisfies the following conditions:*

1. *There exists at least one receiving event r in both Q and V such that $send(r, Q) \neq send(r, V)$.*
2. *Let r be a receiving event in Q and V. If $send(r, Q) \neq send(r, V)$, then $send(r, V)$ must be in $race\_set(r, Q)$.*
3. *Let e be a sending or receiving event in Q. Then, e is not in V if and only if there exists a receiving event r in Q such that $r \in c\text{-}struct(e)$ and $send(r, Q) \neq send(r, V)$.*

Note that the third condition ensures that race variant V is always feasible (i.e., V can be exercised by at least one program execution), regardless of the program's control and data flow. This is because, after the sending partner of a receiving event r is changed, the third condition requires all the events whose existence might be affected by this change to be removed from V. This is a conservative approach since some of the events that are removed may not actually be affected.

TABLE 3
Race Table for Sequence Q_0 in Fig. 1

r_1	r_3
0	1
1	0
1	1

6.2 An Algorithm for Computing Race Variants

Our algorithm for computing race variants builds a "race table" for a given SYN-sequence Q. The race table contains a column for each receiving event in Q whose race set is nonempty. Each row of the race table represents a unique, partially ordered race variant of Q. As an example, Table 3 shows the race table built for sequence Q_0 in Fig. 1. Table 3 has two columns for receive events r_1 and r_3, which are the receive events in Q_0 whose race sets are nonempty. (Note that $race\_set(r_1) = \{s_2\}$ and $race\_set(r_3) = \{s_4\}$.) Variants V_1, V_2, and V_3 in Fig. 1 are derived as described below from rows 1, 2, and 3, respectively, of the race table.

Let r be a receiving event represented by one of the columns and V be a race variant represented by one of the rows. The value v in the row for V and the column for r indicates how r in sequence Q is changed to create variant V:

- $v = -1$ indicates that r is removed from V.
- $v = 0$ indicates that no new sending partner is specified for r in V.
- $v > 0$ indicates that, in V, the sending partner of r is changed to the vth event in $race\_set(r, Q)$, where the sending events in $race\_set(r, Q)$ are arranged in an arbitrary order and the index of the first event in $race\_set(r, Q)$ is 1.

Note that, whenever we change the sending partner of a receiving event r, we also need to remove all the events whose control structure contains r. Also note that the sending partner of a receiving event that does not appear in the table cannot be changed to another sending event.

As an example, for the variant represented by row 1 in Table 3, the value 0 indicates that the send partner of r_1 will be left unchanged, while the value 1 indicates that the send partner of r_3 will be changed to s_4, which is the first (and only) send event in $race\_set(r_3)$. Note that the sending partner of a receiving event could be left unspecified in a variant. For example, in V_2 (derived from row 2), the send partner s_3 of r_3 will be removed because r_1 is in $c\text{-}struct(s_3)$, and the sending partner of r_1 is changed. However, r_3 is not removed as r_1 is not in $c\text{-}struct(r_3)$ and no new send partner is specified for r_3 (the value for r_3 in row 2 is 0). Therefore, the sending partner of r_3 is left unspecified in V_2. In Section 7, we will describe how r_3's send partner is resolved when V_2 is used for prefix-based testing.

A naive algorithm for constructing a race table is as follows: Let $(r_1, r_2, \ldots, r_n)$ be the heading of the race table, which consists of the receiving events whose race sets are nonempty, arranged in an arbitrary order. Let $domain(r_i)$ be the set of values that can appear in the column with heading r_i. If the size of the race set for r_i is denoted as $|race\_set(r_i)|$, then the set of values in $domain(r_i)$ is $\{-1, 0, 1, \ldots, |race\_set(r_i)|\}$. A naive algorithm first generates the set T of tuples that represent all the possible combinations of changes that can be made to all the receiving events in the heading of the race table. These tuples are denoted by

$$T = domain(r_1) \times domain(r_2) \times \ldots \times domain(r_n).$$

The algorithm then adds a tuple $t \in T$ to the race table if t passes a validity check. Denote the individual values in t as $t[1], t[2], \ldots, t[n]$. Then, t is valid if all of the following rules are satisfied:

1. There exists at least one value $t[i]$, $1 \leq i \leq n$, such that $t[i] > 0$.
2. $t[i] = -1$, $1 \leq i \leq n$, if and only if there exists an index j, where $1 \leq j \leq n$ and $j \neq i$, such that $t[j] > 0$ and $r_j \in c\text{-}struct(r_i)$.
3. If $t[i] > 0$, there does not exist an index j, $1 \leq j \leq n$, such that $t[j] > 0$ and $r_j \in c\text{-}struct(s)$, where s is the $t[i]$th sending event in $race\_set(r_i)$.

Note that the first rule implements the first condition in Definition 7. The second and third rules implement the third condition in Definition 7. (The second condition in Definition 7 is reflected in the definition of $domain(r_i)$ above.)

Fig. 7 shows a more efficient algorithm called *Construct-Race-Table*. The gain in efficiency comes from the fact that *Construct-Race-Table* does not generate any invalid tuples. This is in contrast to the naive algorithm, which generates all possible tuples and then filters out the invalid ones. In algorithm *Construct-Race-Table*, the receiving events with nonempty race sets are arranged in left-to-right order with respect to the happened-before relation. (If event a happens before event b, then a appears to the left of b.) Conceptually, a race table is constructed by enumerating the numbers in a number system, where each row in the table is a number in the system and each column is a digit in the number. In the number system, the base of a digit is the size of the race set for the corresponding receiving event plus 1. (Note that each digit may have a different base.) The significance of the digits in a number decreases from left to right.

The rows in the race table are computed iteratively. Starting with the number 0, all the numbers in the number system are enumerated by adding 1 at each iteration. Each new number (not including 0) becomes the next row in the table. Observe that we can add 1 to a number by incrementing the least significant digit g whose value is less than its base minus 1 and setting all the digits that are less significant than g to 0. For example, let 10111 be a binary number (i.e., all the digits in the number have the same base 2). Observe that the second digit (from the left) is the least significant digit whose value is less than 1. In order to add 1 to this number, we increment the second digit from 0 to 1 and set the last three digits to 0. Doing so results in a new number, 11000. There is a slight modification, however, for dealing with the case where the value of a digit is, or becomes, -1, as described below.

To compute the next row in the race table for a SYN-sequence Q, we increment the least significant digit whose value is less than the value of its base minus 1 and whose value is not -1. Let $t[]$ be an array representing the next row

```
RaceTable Construct-Race-Table (Q: a SYN-sequence) {
1. initialize table = (heading, rows) to be an empty race table;
2. R = {r ∈ Q | |race_set(r)| > 0};
3. let heading = (r1, r2, ..., r|R|} be a topological order of R w.r.t the happened-before relation;
4. D = {d1, d2, ..., d|R|}, where di = |race_set(ri)|;
5. let t be an array of length |R| and initialize t with all 0s;
6. while (true) {
7.   find the largest index i such that t[i] < di and t[i] != -1;
8.   if (such an index i does not exist)
9.       break;
10. t[i]++;
11. if (t[i] == 1)  // just changed t[i] from 0 to 1
12.      for (i < j ≤|R|)
13.          if (t[j] ≠-1 and ri ∈ c-struct(rj))
14.              t[j] = -1;
15. for (i < j ≤|R|)
16.      if (t[j] == dj) {
17.          t[j] = 0;  // just changed t[j] from dj to 0
18.          for (j < k ≤|R|)
19.              if (t[k] == -1 and rj ∈ c-struct(rk) and there is no index l, 1≤ l < k, such that t[l] > 0 and rl ∈ c-struct(rk))
20.                  t[k] = 0;
21.      } // end if
22. let s be the t[i]th sending event in race_set(ri);
23. if (there does not exist an index j, 1 ≤j≤|R|, such that t[j] > 0 and rj ∈ c-struct(s))
24.      add t to rows;
25.} // end while
26. return table;
}
```

Fig. 7. Algorithm *Construct-Race-Table*.

in the race table. We use the following rules to ensure that $t[]$ represents a valid race variant V of sequence Q:

1. Whenever we change $t[i]$ from 0 to 1, which means that the sending partner of r_i will be changed, we set $t[j] = -1$, $i < j \leq n$, if $r_i \in c\text{-}struct(r_j)$ (lines 13 and 14). This is to remove r_j from V as changing the sending partner of r_i may affect the existence of the events whose control-structures contain r_i.
2. Let d_j be the base of digit $t[j]$ minus 1, which is also the size of r_j's race set. Whenever we change $t[j]$ from d_j to 0, which means that the sending partner of r_j will be changed back to its original sending partner in Q, we set $t[k] = 0$, $j < k \leq n$, if the current value of $t[k]$ is -1 and there no longer exists an index l, $1 \leq l < k$, such that $t[l] > 0$ and $r_l \in c\text{-}struct(r_k)$ (lines 18, 19, and 20). In other words, if r_j is the only event causing $t[k]$ to be set to -1 (due to the application of rule 1) and if we change r_j's sending partner back to its original sending partner in Q, then we change $t[k]$ from -1 to 0 so that r_k is no longer removed from the variant.
3. Assume that we have incremented $t[i]$. Let s be the $t[i]$th sending event in $race\_set(r_i)$. We need to check whether there exists an index j such that $t[j] > 0$ and $r_j \in c\text{-}struct(s)$. Array $t[]$ is added to the race table as the next row if and only if such an index j does not exist (lines 23 and 24). This is because if such an index j does exist, s should be removed from the variant and, thus, $t[]$ would not represent a valid variant.

Notice that when we change $t[i]$ from 0 to 1 or from $|d_i|$ to 0, we need only check the values of $t[j]$, $i < j \leq n$. This is because receiving events are ordered from left-to-right based on the happened-before relation. This ordering also ensures that the value represented by $t[]$ increases at each iteration. Therefore, algorithm *Construct-Race-Table* will eventually terminate.

As an example, consider how to add 1 to the number represented by the first row (0, 1) in Table 3. In line 10 of *Construct-Race-Table*, we increment the first (i.e., leftmost) column from 0 to 1 since the first column is the least significant column whose value is less than its base minus 1 and is not -1. (The heading for the first column is r_1 and the base for the first column is 2, which is $|race\_set(r_1)| + 1$.) Note that changing the first column does not cause the second column to be changed to -1 in lines 13 and 14 since $r_1 \notin c\text{-}struct(r_3)$ (where r_3 is the heading for the second column). In the for-loop starting at line 15, we check the value in the second column. The check in line 16 shows that the current value of the second column equals its base value minus 1, so the second column is set to 0 in line 17. This ends the for-loop of line 15. Finally, lines 22 and 23 check the new send partner s_2 of r_1. (Note that the new value for the first column is 1 and the first and only send event in $race\_set(r_1)$ is s_2.) Since s_2 does not happen after any receive event whose send partner has been changed, there is no reason to remove s_2, which means that the variant is valid. Therefore, at line 24, we add the new number (1, 0) into the table as the next row.

We wish to stress that algorithm *Construct-Race-Table* guarantees that every row in a race table represents a

unique, partially ordered variant. Let V_1 and V_2 be the race variants represented by any two rows in the table. Algorithm *Construct-Race-Table* ensures that there exists at least one receiving event r such that r appears in both V_1 and V_2, but the sending partner of r in V_1 is different from that in V_2. This ensures that V_1 and V_2 are different partial orders. Therefore, our algorithm deals with partial orders directly—it never generates any interleavings that are only different in the ordering of some independent events.

Finally, we consider the time complexity of algorithm *Construct-Race-Table*. Let R be the set of receiving events whose race sets are nonempty. The algorithm is dominated by the while-loop, which is further domained by the inner double for-loop (lines 15-21). Note that every element $t[j]$, $i < j \leq |R|$, can be visited at most once in the double for-loop. Thus, the time complexity of the double for-loop (lines 15-21) is $O(|R|^2)$. Considering that each iteration of the while-loop usually adds one row to the race table, the time complexity of the algorithm is $O(|R|^2 * |V|)$, where V is the set of race variants for Q.

7 A Reachability Testing Algorithm

A reachability testing algorithm drives the reachability testing process by collecting SYN-sequences, generating variants, and performing prefix-based testing with the variants. In order to reduce test effort while maximizing test coverage, it is desirable to exercise every (partially ordered) SYN-sequence *exactly* once during reachability testing. However, if a newly derived race variant V is a prefix of a SYN-sequence Q that has already been exercised, then prefix-based testing with V could exercise Q again. To deal with this potential duplication problem, all the existing reachability testing algorithms need to save the history of SYN-sequences that have already been exercised. A newly derived variant is used for prefix-based testing only if it is not a prefix of any SYN-sequence in the history. For large and/or complex programs, the cost of saving the history can be prohibitive both in terms of the space to store the history and the time to search it. In this section, we present a new reachability testing algorithm that does not save any SYN-sequences, but still guarantees that every SYN-sequence will be exercised exactly once. We note that the number of test runs performed by our algorithm may be slightly larger than the number of SYN-sequences that can be exercised by the program under test, which will be discussed later.

7.1 A Graph-Theoretic Perspective

To understand our new reachability testing algorithm, we consider the reachability testing problem from a graph-theoretic perspective. Let CP be a concurrent program. All the possible SYN-sequences that can be exercised by CP with input X can be organized into a directed graph G, which we refer to as a *Sequence/Variant* graph or, simply, an *S/V*-graph. Each node n in G is labeled by a SYN-sequence $seq(n)$ that can be exercised by CP with input X. An edge e from node n to node n' is labeled by a race variant $var(e)$ of $seq(n)$, indicating that $seq(n')$ could be exercised by a prefix-based test run with $var(e)$. Note that node n may have more than one outgoing edge that is labeled by the same variant

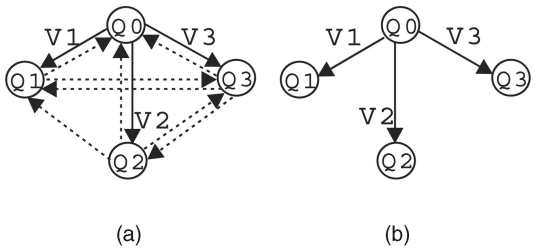

Fig. 8. (a) The S/V-graph for the example program in Fig. 1 and (b) a spanning tree for the graph.

of $seq(n)$. This is because prefix-based testing with a race variant forces the variant to be exercised at the beginning of the test run and then lets the run continue *nondeterministically*, where the nondeterministic portion is not controlled and can exercise different sequences in different test runs.

Theorem 1. *Let CP be a concurrent program. Let G be the S/V-graph of CP with input X. Then, G is strongly connected.*

The proof of Theorem 1 is provided in the supplementary material, which can be found on the Computer Society Digital Library at http://computer.org/tse/archives.htm.

From a graph-theoretic perspective, the goal of reachability testing is to construct a spanning tree of the S/V-graph of a concurrent program with a given input. (Of course, reachability testing does not actually construct a spanning tree, but the sequences exercised during reachability testing and the variants used to collect these sequences should represent a spanning tree.) Note that a spanning tree of S/V-graph G is a subgraph of G that is a tree (i.e., a graph with no cycles) and that connects the n nodes of G with $n-1$ edges (i.e., each node, except the root, has one and only one incoming edge.) Since an S/V-graph is strongly connected, reachability testing can start from an arbitrary node. This explains why we can start reachability testing with the SYN-sequence collected from a nondeterministic test run. Note that each race variant is used to conduct a single test run during reachability testing. Therefore, in a spanning tree that represents the reachability testing process, no two edges are labeled with the same variant.

Fig. 8a shows the S/V-graph for the example program in Fig. 1a and Fig. 8b shows a spanning tree that represents the application of reachability testing (Fig. 1b) to this program. Due to the space constraints, the race variants represented by the dashed edges in the S/V-graph are not shown. Note that, in the S/V graph, there is an edge from Q_2 to Q_1, but no edge from Q_1 to Q_2. Also, note that other spanning trees are possible since the reachability testing process is inherently nondeterministic.

Recall that existing reachability testing algorithms must save the history of SYN-sequences to avoid exercising duplicate sequences. From a graph-theoretic perspective, the purpose of saving this test history is to avoid generating the same node more than once during the construction of a spanning tree. Therefore, the main challenge for our new algorithm is to avoid generating the same node (i.e., exercising the same sequence) more than once *without* saving the list of the nodes that have already been

generated. Obviously, this challenge cannot be solved purely as a graph problem. Instead, it must be solved within the context of reachability testing.

7.2 Path Constraints

Let G be the S/V-graph of a concurrent program CP with input X. The main idea of our new algorithm is as follows: If we can find some constraints on the paths in G such that, given two arbitrary nodes n and n' in G, there is exactly one acyclic path from n to n' that satisfies these constraints, then we can construct a spanning tree of G by enforcing these constraints, i.e., by only generating paths that satisfy these constraints. In this section, we will define two such path constraints. In the next section, we will show how they are enforced in our new algorithm.

We first introduce the notion of a race difference, which is needed to define our path constraints.

Definition 8. *Let Q and Q' be two sequences. Let r be a receiving event that exists in both Q and Q'. There is a race difference with r if $send(r, Q)$ and $send(r, Q')$ are both defined, but $send(r, Q) \neq send(r, Q')$.*

Recall that we assume that the only source of nondeterminism is due to concurrency, i.e., the order in which threads synchronize and communicate. Under this assumption, any difference between two sequences (of the same program with the same input) can be traced back to a race difference. This property is formally stated in Lemma 1. In the rest of the paper, we will only be interested in race differences and will refer to a race difference simply as a difference unless otherwise specified.

Lemma 1. *Let CP be a concurrent program. Let Q and Q' be two SYN-sequences of CP with input X. Then, Q and Q' are equal if and only if they have no race difference.*

The proof of Lemma 1 is provided in the supplementary material, which can be found on the Computer Society Digital Library at http://computer.org/tse/archives.htm.

Let n and n' be two arbitrary nodes in G. Let H be a path from node n to node n', i.e., $H = n_1 e_1 n_2 e_2 \ldots n_m$, where $n = n_1$ and $n' = n_m$. We consider each edge e_i along path H to represent a transformation of node n_i into node n_{i+1}. This transformation is realized by changing the sending partner of one or more receiving events in $seq(n_i)$ to derive the race variant $var(e_i)$ of $seq(n_i)$ and then performing prefix-based testing with variant $var(e_i)$ to collect $seq(n_{i+1})$. The path constraints to be defined impose restrictions on how the sending partner of a receiving event can be changed by such a transformation. We will say that the sending partner of a receiving event r is *explicitly* changed by edge e_i if r's sending partner is changed to a different sending event in $var(e_i)$. This is distinguished from the case where the sending partner s of a receiving event r is removed from $var(e_i)$ (due to the explicit change of another receiving event in *c-struct*(s)) and then r is synchronized with a different sending event during prefix-based testing with $var(e_i)$. We will say that edge e_i reconciles a difference between $seq(n_i)$ and $seq(n')$ if the sending partner of a receiving event r in $seq(n_i)$ is *explicitly* changed by e_i to match the sending partner of r in $seq(n')$. The two path constraints are defined below:

C1. The sending partner of a receiving event can be explicitly changed by edge e_i only if such a change reconciles a (race) difference between $seq(n_i)$ and $seq(n')$.

Constraint C1 prevents any edge e_i from introducing any new difference between $seq(n_i)$ and $seq(n')$. This ensures that H is an acyclic path. To the contrary, assume that H is a cyclic path, i.e., $n = n'$. There must exist at least one receiving event r in $seq(n)$ such that r's sending partner s is explicitly changed to another send event s', where $s \neq s'$, by edge e_1. (Recall that e_1 is derived by changing the outcome of one or more race conditions of $seq(n)$.) The change from s to s' is forbidden by C1 because $n = n'$ and such a change would create, rather than reconcile, a difference between $seq(n)$ and $seq(n')$.

For example, the S/V graph in Fig. 8 contains a cycle $Q_0Q_1Q_0$. Note that, for this path, $seq(n)$ and $seq(n')$ in constraint C1 both refer to sequence Q_0. Note also that the first edge in the cycle, which is from Q_0 to Q_1 and is labeled by V_1, changes the sending partner of r_3 from s_3 to s_4. This cycle does not satisfy constraint C1 as the change creates a new difference for r_3 between Q_0 and itself. Therefore, this cycle is excluded from the spanning tree.

C2. Each edge e_i must reconcile all the (race) differences between $seq(n_i)$ and $seq(n')$.

Observe that, for any node n_i, there exists at most one variant of $seq(n_i)$ that reconciles all the differences between $seq(n_i)$ and $seq(n')$ and does not create any new difference. Therefore, C2 can be satisfied by at most one path from n to n'.

As an example, consider the acyclic path $Q_0Q_2Q_3$ in the S/V-graph in Fig. 8. Observe that receive events r_1 and r_3 exist in both Q_0 and Q_3, but their sending partners in Q_0 are different from their sending partners in Q_3 (see Fig. 1). Also observe that the first edge V_2 along path $Q_0Q_2Q_3$ reconciles the difference with r_1 but not the difference with r_3. This violates constraint C2, which requires each edge to reconcile all the race differences. As a result, path $Q_0Q_2Q_3$ is excluded from the spanning tree. Note that path Q_0Q_3 is the only path from Q_0 to Q_3 that satisfies constraints C1 and C2 and is thus included in the spanning tree. The only edge V_3 in path Q_0Q_3 reconciles the differences for both r_1 and r_3.

Based on the above discussion, C1 and C2 can be satisfied by at most one acyclic path from n to n'. It can also be shown that C1 and C2 can be satisfied by at least one acyclic path from n to n'. Note that the proofs in the supplementary material provide a formal justification that C1 and C2 can be satisfied by exactly one acyclic path from n to n'.

7.3 The Algorithm

Fig. 9 shows our new reachability testing algorithm. Algorithm *Reachability-Testing* starts with the SYN-sequence Q_0 collected from a nondeterministic test run (line 2). Recall that a nondeterministic test run is a test run in which nondeterminism is resolved arbitrarily, i.e., without controlling which SYN-sequence gets exercised. It then uses function *GenerateVariants* in Fig. 10 to generate a set of race variants of Q_0 (line 4). Note that *GenerateVariants* is called with the *empty* variant, which reflects the fact that a nondeterministic test run can be considered as a prefix-based test run with a prefix that contains no events. Each of these race variants is used to

```
ALGORITHM Reachability-Testing (CP: a concurrent program; X: an input of CP) {
1.  let variants be an empty set;
2.  collect a SYN-sequence Q0 by executing CP with input X non-deterministically;
3.  let V0 be the special empty variant that contains no events
4.  variants = GenerateVariants(Q, V0)
5.  while (variants is not empty) {
6.    withdraw a variant V from variants;
7.    collect a SYN-sequence Q by conducting a modified prefix-based test run with V;
8.    variants = variants ∪ GenerateVariants(Q, V);
  }
}
```

Fig. 9. Algorithm *Reachability-Testing*.

```
FUNCTION GenerateVariants (SYN-sequence Q, Variant V) {
// sequence Q was collected during prefix-based testing with variant V
    // prune "old" sending events
1.  for each receiving event r in V (and thus in Q too)
2.     race_set(r, Q) = race_set(r, Q) - race_set(r, V)

    // generate a subset of the race variants of Q
3.  Use algorithm Construct-Race-Table to construct a race table with statement 2 in
      the algorithm replaced with the following statement
       R = {r ∈ Q | |race_set(r)| > 0 and r.color = white}
4.  Derive a list variants(Q) of race variants, one variant from each row of the race table

    // set the colors of the receiving events in the variants of Q
5.  for (each variant V' in variants (Q)) {
6.    for (each receiving event r in V' whose sending partner was explicitly changed) {
7.      r.color = black;
8.      for (each receiving event r' that happens before r in V') {
9.        r'.color = black;
10.     }
11.   }
12. }
13. return variants (Q);
}
```

Fig. 10. Function *GenerateVariants*.

collect a new SYN-sequence by performing prefix-based testing, but with a slight modification to handle the case where the sending partner of a receiving event is not specified (line 7). This modification will be discussed later. Each newly collected SYN-sequence is then used to derive new race variants (line 8). This procedure is repeated until no more race variants can be generated. We stress that algorithm *Reachability-Testing* does not save any SYN-sequences that have already been exercised.

Given a SYN-sequence Q and the race variant V that was used to collect Q, function *GenerateVariants* generates only a subset of the variants of Q (instead of all the variants of Q). In *GenerateVariants*, the receiving events in Q and V are colored either white or black. The color of a receiving event r in V is inherited by the same event r in Q. (Recall that V is a prefix of Q, so each event in V is also in Q.) Receiving events that are in Q but not in V are colored white. The color of a receiving event r in Q restricts how the sending partner of r can be changed to derive race variants of Q: *white* indicates that the sending partner of r can be explicitly changed; *black* indicates that the sending partner of r cannot be explicitly changed. The color of a receiving event can change from white to black, but never from black to white.

Next, we explain how algorithm *Reachability-Testing* enforces the two path constraints C1 and C2. In previous subsections, we related the reachability testing problem to the problem of constructing a spanning tree of an S/V-graph. Constraints C1 and C2 are constraints on the paths that can be taken through an S/V-graph from an arbitrary node n to another n'. During the actual reachability testing process, however, there is no such S/V-graph since we do not know a priori which SYN-sequences the program under test can exercise. Therefore, the main technical challenge of our algorithm is *how to enforce constraints C1 and C2 without a priori knowledge about the S/V-graph and/or any particular nodes n and n'*. For each constraint, we will describe a strategy for enforcing the constraint and show how this strategy is implemented in algorithm *Reachability-Testing*.

C1. The sending partner of a receiving event can be explicitly changed by edge e_i only if such a change reconciles a (race) difference between $seq(n_i)$ and $seq(n')$.

We enforce this constraint using the following strategy: After the sending partner s of a receiving event r in a sequence Q is explicitly changed to another sending event s', we will ensure that both r and s' exist and remain

synchronized with each other in any sequence Q' exercised afterward. Since $send(r, Q) = s$, and $send(r, Q') = s'$, the change from s to s' actually reconciles a difference between Q and Q'. This is equivalent to saying that the sending partner of r in a sequence Q can be explicitly changed along a path only if such a change reconciles a difference between Q and the sequence Q' that is reached at the end of the path.

The implementation of this strategy consists of two parts:

1. If the sending partner s of a receiving event r in Q is explicitly changed to another sending event s' to derive a race variant V', then the color of r in V' is set to black in line 7 of function *GenerateVariants*. Line 3 of function *GenerateVariants* excludes black receiving events from the heading of the race table constructed by algorithm *Construct-Race-Table* (even though black receiving events may have nonempty race sets). Since the color of an event never changes from black to white, this prevents the sending partner s' of r from being explicitly changed afterward.
2. When we set the color of r to black, we also set the color of a receiving event r' to black if r' happens before r in V' (see lines 8 and 9 in function *GenerateVariants*). Doing so prevents the sending partner of any receiving event like r' from being changed and, thus, ensures that r, as well as the new sending partner s' of r, must exist in any sequence that is exercised afterwards. (Recall from Definition 7 condition (3) that we remove an event e if and only if we change the sending partner of a receiving event in e's control-structure to another sending event. Since all of the events that happen before r, which includes all of the events in the control-structures of r and s', are now colored black and, hence, cannot be changed, r and s' can never be removed.)

Earlier, we showed that the cyclic path $Q_0Q_1Q_0$ in the S/V-graph in Fig. 8 does not satisfy constraint C1. Note that variant V_1 (the portion of Q_1 that is above the dashed line) was derived by changing the sending partner of r_3 from s_3 to s_4. Therefore, the color of r_3 in V_1 will be black and this color will be inherited by r_3 in Q_1. Thus, r_3 will be excluded from the heading of the race table for Q_1. As a result, the sending partner of r_3 cannot be explicitly changed again when we derive the race variants of Q_1. Note that this prevents the cyclic path $Q_0Q_1Q_0$ from being generated during reachability testing.

C2. Each edge e_i must reconcile all the (race) differences between $seq(n_i)$ and $seq(n')$.

Our strategy for enforcing C2 is as follows: If there are differences that can be reconciled by an edge but are not reconciled by that edge, then we make sure that these differences cannot be reconciled afterward. Therefore, if a path contains an edge e_i that does not reconcile some of the differences between $seq(n_i)$ and $seq(n')$, then these differences will never be reconciled along the path. As a result, the path will not be generated during reachability testing.

The above strategy is implemented in *GenerateVariants* by removing "old" sending events from the race sets of "old"

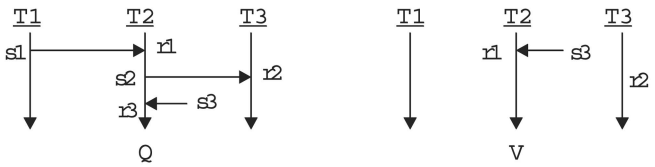

Fig. 11. Modified prefix-based testing.

receiving events (lines 1 and 2). An event in a SYN-sequence Q is old if it also appeared in the variant V that was used to collect Q. Note that the difference with a receiving event r can be reconciled by an edge e_i only if the sending partner s of r in $seq(n')$ exists in $seq(n_i)$. Therefore, if the difference with r was not reconciled by e_i, then both s and r will become "old" events in any sequence that can be reached afterward if they exist in the sequence. Hence, s will always be removed from the race set of r, preventing the difference with r from ever being reconciled afterward.

Earlier, we showed that path $Q_0Q_1Q_3$ in the S/V-graph in Fig. 8 does not satisfy constraint C2. Note that events r_1 and s_2 in Q_1 are old events because they appear in both V_1 (the variant used to collect Q_1) and Q_1. In function *GenerateVariants*, s_2 will be removed from the race set of r_1 in Q_1, which means that we will not be able to change the sending partner of r_1 to s_2 when we derive the race variants of Q_1. As a result, path $Q_0Q_1Q_3$ cannot be generated as, in order to reach Q_3 from Q_1, we would have to change the send partner of r_1 from s_1 to s_2.

There is a special case of C2 that must be handled. Consider sequence Q in Fig. 11. Suppose we change the sending partner of receiving event r_1 from s_1 to s_3 to derive a variant V. Since r_1 is in the control structure of the sending partner s_2 of receiving event r_2 but r_1 is not in the control structure of r_2, sending event s_2 will be removed from V but r_2 will stay in V. If no new sending partner is specified for r_2, r_2 will have its sending partner left unspecified in V.

We call a receiving event an "unmatched" receiving event if its sending partner is not specified in a variant. During prefix-based testing with a variant that contains unmatched receiving events, additional runtime control is needed to ensure that the unmatched events are not synchronized with any sending events that already appear in the variant. This is to ensure that "old" receiving events are not synchronized with "old" sending events. As an example, consider V_2 in Fig. 1, where the sending partner of r_3 is not specified. (Note that send event s_3, which is the original sending partner of r_3 in Q_0, is removed from V_2 and no other sending partner is specified for r_3 in V_2.) During prefix-based testing with V_2, we will ensure that r_3 does not receive the message sent by s_4 (which is an old send event) as, otherwise, we would exercise Q_3 twice.

In the case where an unmatched receiving event can only be synchronized with an "old" sending event, the test run will not be allowed to complete. Instead, the test run will be allowed to proceed up to the point where the test run could not continue unless we were to allow some unmatched receiving events to be synchronized with old sending events. The sequence exercised by such a partial test run is collected (including the unmatched receiving events) and

the reachability testing process continues normally. We point out that such a partial test run does not represent a deadlock in the application since the run could continue if unmatched receiving events were allowed to be synchronized with "old" sending events. On the other hand, if an unmatched receiving event cannot be synchronized with any sending event at all, then the application is deadlocked or livelocked and reachability testing will terminate. Our prototype tool RichTest issues a timeout after observing for a specified period of time in which the application has neither exercised any events nor terminated, and then decides whether the reachability testing process should be continued or terminated, as described above. Note that, since partial test runs do not represent complete SYN-sequences, the number of test runs we perform during reachability testing may sometimes be larger than the number of SYN-sequences that can be exercised by the program under test. However, we expect the number of partial test runs to be small. This is evidenced by our case studies in which only one partial test run was generated for all the programs, as reported in Section 8.

Now, we consider the time complexity of algorithm *GenerateVariants*, which is dominated by lines 3 and 4. As shown earlier, the original *Construct-Race-Table* algorithm is in $O(|R|^2 * |V|)$, where $|R|$ is the number of receiving events in the race table heading and $|V|$ is the number of race variants of Q. The modification required for removing black receiving events from the header of the race table does not change the time complexity. In line 4, each time we change the sending partner of a receiving event r, we need to remove all the events whose control structures contain r. The complexity of line 4 is $O(|E| * |V|)$, where $|E|$ is the total number of sending and receiving events in Q. Therefore, the complexity of function *GenerateVariants* is bounded by $O(|E|^2 * |V|)$. It follows that the time complexity of algorithm *Reachability-Testing* is $O(n * |E_{max}|^2 * |V_{max}|)$, where n is the number of possible SYN-sequences, $|E_{max}|$ is the maximum number of events in a SYN-sequence, and $|V_{max}|$ is the maximum number of variants for a SYN-sequence. The space complexity of algorithm *Reachability-Testing* is dominated by the size of set variants. Note that we can encode the variants of a SYN-sequence efficiently using the race table of the sequence. If set variants is implemented as a stack, its size is $O(D)$, where D is the maximum depth of the spanning tree explored by algorithm *Reachability-Testing*.

The following theorem states the correctness of algorithm *Reachability-Testing*.

Theorem 2. *Given a concurrent program CP and an input X, algorithm Reachability-Testing exercises every partially ordered SYN-sequence of CP with input X exactly once.*

The formal proof of Theorem 2 is provided in the supplementary material, which can be found on the Computer Society Digital Library at http://computer.org/tse/archives.htm.

As discussed in Section 5.1, SYN-sequences that represent different partial orders can be equivalent. This suggests that it is not always necessary to exercise every partially ordered SYN-sequence. Below, we describe a reduction, which we will refer to as the P/V reduction, for programs that use counting semaphores. The P/V reduction is implemented in our prototype and can be optionally enabled to further reduce the number of sequences exercised during reachability testing.

The P/V reduction consists of ignoring a race between P and V operations on a counting semaphore s in the following two cases:

- If two $s.V$ operations are called concurrently, the race between the two calls to $s.V$ can be ignored.
- If an $s.V$ operation and an $s.P$ operation are called concurrently (i.e., neither one happens before the other) and the call to $s.V$ is completed before the call to $s.P$, the race between the two calls to $s.V$ and $s.P$ can be ignored.

Note that, by ignoring a race, we mean that no race variants are generated to change the outcome of the race. To see why the P/V reduction is safe, observe that two V operations are independent in any state and a V operation is independent with a P operation in any state where the value of the semaphore is greater than 0. (Two operations are independent in a state g if they do not enable or disable each other in g and executing them from g in any order will reach the same successor state.) Ignoring the race between a V operation and another operation X (which can be V or P) that completes after the V operation means that those sequences in which operation X is completed before the V operation will not be exercised during reachability testing. Let Q be such a sequence that is not exercised due to the P/V reduction on a counting semaphore s. We will show that Q is equivalent to a sequence Q' that is guaranteed to be exercised. Intuitively, two sequences are equivalent if they are guaranteed to produce the same outcome. Since the reduction is applied to completed P and V operations, we are concerned with the order in which P and V operations are completed on semaphore s. Let $t_1 t_2 \ldots t_n$ be the (totally ordered) sequence of semaphore-completion events on s in Q. (Note that, since all incomplete P and V operations are concurrent and never happen before a completed operation, they can be placed at the end of a totally ordered sequence and are thus ignored in our discussion.) Assume that t_i is operation X and t_j is the concerned V operation, where $1 \leq i < j < n$. Note that if a P operation is completed before a V operation, the value of the semaphore must have been greater than 0 when the P operation was started. Thus, t_j is independent of t_k, $1 \leq k < n$, at each state s_k where t_k was executed Therefore, t_j can always be moved toward the beginning of a sequence, which means that there must exist a sequence Q' that is equivalent to Q and is guaranteed to be exercised. We point out that this reduction is also used to compute persistent sets in partial order reduction [13].

Note that the above reduction has no effect on a counting semaphore that is used solely to create critical sections. To see this, let counting semaphore *mutex* be such a semaphore. *Mutex* is initialized to 1 and is used in the following way: $mutex.P(); < criticalsection; > mutex.V()$. Since at most one thread can enter a critical section at any time, there can never be concurrent calls to $mutex.V$ operations

TABLE 4
Empirical Results

Program	Seqs		Program	Seqs		Program	Seqs
BB-select	144		RW-select	768		DP-monitorSU (3P)	30
BB-semaphore	324		RW-semaphore	21744		DP-monitorSU (4P)	624
BB-monitorSU	720		RW-monitorSU	13320		DP-monitorSU (5P)	19330
BB-monitorSC	12096		RW-monitorSC	61716		DME	4032

nor can there ever be concurrent calls to $mutex.P$ and $mutex.V$ operations.

Note also that the reduction does not apply to binary semaphores. The reason is twofold. First, as defined in Section 2, P and V operations on a binary semaphore can enable or disable each other and are thus never independent. Second, two operations on a binary semaphore never race with a V operation since only one of them can be completed at any given time.

8 Empirical Results

We implemented our reachability testing algorithms in a prototype tool called RichTest. RichTest is developed in Java and consists of three main components: a synchronization library, a race variant generator class, and a test driver class. The synchronization library provides classes for simulating semaphores, monitors, and message passing with selective waits. The synchronization classes contain the necessary control for tracing SYN-sequences and replaying variants. The race variant generator implements function *GenerateVariants*, i.e., it inputs a SYN-sequence and generates a subset of the race variants of the sequence. The test driver is responsible for coordinating the exchange of variants and SYN-sequences between the synchronization classes and the variant generator. These three components and the application form a single Java program that performs the reachability testing process.

We wish to stress that RichTest does not require any modifications to the JVM or any modifications to, or direct interactions with, the operating system. Instead, the synchronization classes contain the necessary control for reachability testing. In trace mode, the synchronization classes record synchronization events at appropriate points and assign timestamps to these events. In replay mode, the synchronization classes implement the replay techniques that have been developed for the various constructs. We are applying this same approach to building portable reachability testing tools for multithreaded C++ programs that use thread libraries in Windows, Solaris, and Unix.

As a proof-of-concept, we conducted an empirical study in which RichTest was used to perform reachability testing on several programs. The programs used as objects of the study include:[1]

1. BB—a solution to the bounded-buffer problem where the buffer is protected using either a selective wait, semaphores, an SU monitor, or an SC monitor. Program BB had three producers and three consumers and a buffer with two slots [9].
2. RW—a solution to the readers and writers problem using a selective wait, semaphores, an SU monitor, or an SC monitor. Program RW had three readers and two writers [9].
3. DP—a solution that uses an SU monitor to solve the dining philosophers problem without deadlock or starvation [9]. Program DP had three, four, or five philosophers.
4. DME—a solution to the distributed mutual exclusion problem with three processes and two threads per process [25]. Each process has one thread that accesses the critical section and another thread that helps with message processing. This program uses message passing for communication between processes and semaphores for synchronization between threads within a single process.

Note that these solutions represent synchronization/communication patterns that are commonly found in many practical applications. Also note that the synchronization behavior of all the above programs is independent from their inputs. We stress that the complexity of the synchronization behavior of a program mainly depends on the way in which the threads communicate and synchronize, not the size of the program.

Table 4 summarizes the results for all the programs. For each program, we show the number of sequences exercised during reachability testing. For all of the programs in Table 4, the number of sequences exercised equaled the number of test runs, i.e., there were no incomplete test runs. Also, we note that the time needed to perform these test runs largely depends on the program under test and is proportional to the number of sequences that have to be exercised.

Recall that previous reachability testing algorithms required the history of sequences to be saved and searched. Each sequence that is collected is added to the history and each variant generated is compared to all the sequences in the history. If n sequences are exercised during reachability testing, the total number of searches is $1 + 2 + \ldots + n = n(n+1)/2$. To shed some light on the performance improvement created by removing the need to save the test history, the time and space needed for testing BB-monitorSC was 3 minutes and 24 seconds and 0.4 MBs when the test history was not saved and 5 minutes and 56 seconds and 16.5 MBs when the test history was saved. Note that, for BB-monitorSC, our new algorithm avoids over 73 million comparisons that would

1. These programs can be obtained by contacting the authors.

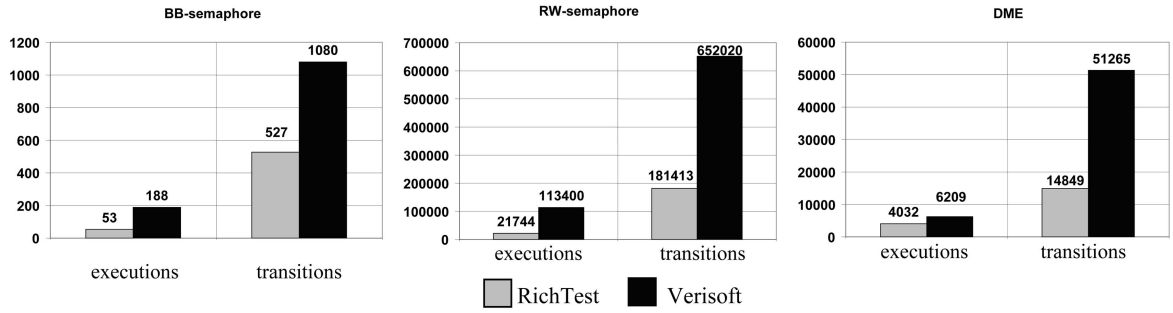

Fig. 12. Comparison between RichTest and VeriSoft for BB-semaphore, RW-semaphore, and DME programs.

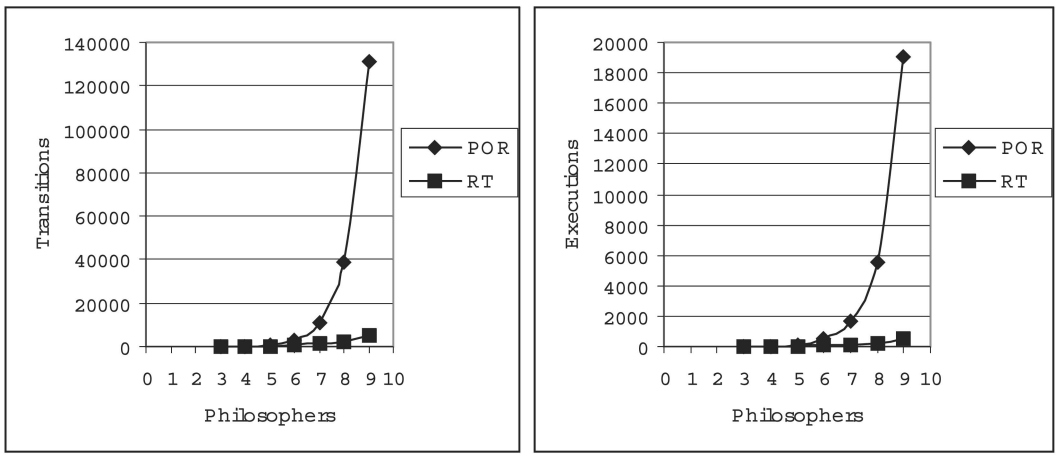

Fig. 13. Comparison between RichTest and VeriSoft for the dining philosophers problem.

otherwise need to be searched through the test history. Reachability testing with RW-semaphore, RW-monitorSU, and RW-monitorSC ran out of memory when the test history was saved on a system with 512 MB RAM. When the test history was not saved, none of these three programs required more than 4.7 MB RAM.

Reachability testing was performed on the three programs involving semaphores, namely, BB-semaphore, RW-semaphore, and DME, with and without the P/V reduction described at the end of Section 7. Program BB-semaphore exercised 324 sequences without the P/V reduction and 53 sequences with the P/V reduction, where 53 was the average number of sequences exercised over 10 test runs. (Note that the number of sequences that are exercised when the P/V reduction is applied depends on the types of P/V races that occur and may vary for different test runs of the same program with the same input.) The P/V reduction did not reduce the number of sequences exercised by RW-semaphore and DME. As we mentioned in Section 7, the P/V reduction does not affect P and V operations that are used to create critical sections. All three programs contain critical sections. However, unlike RW-semaphore and DME, BB-semaphore contains P and V operations that are used for conditional synchronization. These P and V operations are executed outside the critical section and, so, the reduction does affect these operations. This accounts for the difference in the reduction results.

In order to compare reachability testing to partial order reduction, we compared the performance of RichTest and a partial order reduction-based tool called VeriSoft from Bell Labs [13]. As explained in Section 9, VeriSoft is a tool that is very similar to RichTest, except that VeriSoft uses partial-order reduction to avoid exercising redundant interleavings. Both techniques were applied to the semaphore-based programs, namely, BB-semaphore and RW-semaphore, and to the message passing program DME. (VeriSoft supports the use of semaphores and FCFS message buffers, but not monitors or select statements.) The results for the three programs are shown in Fig. 12. (The RichTest results were generated using the P/V reduction; Verisoft also uses this reduction.) These results indicate that RichTest performs significantly better than VeriSoft for these programs in terms of the number of times the programs are executed and the number of transitions that are explored. Note that the fault detection effectiveness of RichTest and VeriSoft are, in principle, the same as both techniques can explore all the SYN-sequences of a concurrent program.

In [13], Godefroid reported the results of applying VeriSoft to a dining philosophers program that contained a deadlock. The program was executed with three to nine philosophers. We rewrote the same program in Java and applied reachability testing to the Java program. Fig. 13 shows our results. The graph on the left in Fig. 13 shows the number of transitions explored by VeriSoft and by RichTest for various numbers of philosophers. In the case of nine philosophers, VeriSoft explored 131,478 transitions, while RichTest explored only 5,004 transitions. The graph on the right in Fig. 13 shows the numbers of times the program was executed to perform the selective search and to perform reachability testing. In the case of nine philosophers,

VeriSoft required 19,023 executions, whereas RichTest required only 511 executions. The results indicate that RichTest performs significantly better than VeriSoft for this program. We point out that one test run with four philosophers was incomplete and collected a partial sequence. This was the only time a partial sequence was collected in all our case studies.

Note that, in order to reduce the number of transitions that are explored, VeriSoft allows the user to manually enter structural properties of the form "operation x is allowed to be executed on semaphore/message buffer y by process z." We specified these properties for the case study programs. Since the running times of both techniques are proportional to the number of explored transitions and the number of executions, the results of this study show that reachability testing may perform significantly faster than partial order reduction. It is hard to be more specific about the running times of RichTest and VeriSoft. For program *RW-semaphore* in Fig. 12, VeriSoft ran for 48 minutes, while RichTest ran for 17 minutes. However, VeriSoft works on multiprocess C programs and RichTest works on multithreaded Java programs, which affects the execution times. We are porting RichTest to C/C++, which will allow a better comparison to be made.

Since reachability testing can exercise every SYN-sequence, it guarantees the detection of deadlocks and assertion violations. An assertion in a process is a Boolean expression that can test and compare the value of variables and data structures local to the process. In practice, the ability of reachability testing to find faults depends on the techniques that are used for evaluating test results and detecting program failures and on whether reachability testing is exhaustive or nonexhaustive. Some failures, like deadlock, are impossible to miss, whereas others may only be captured by a particular assertion or property that may or may not be among those checked by the user. Exhaustive reachability testing enables better fault detection since it ensures, when practical, that all the SYN-sequences of a concurrent program with a given input are covered. When exhaustive testing is not practical, reachability testing can still enable better test coverage. Reachability testing is more efficient than random, nondeterministic testing since each execution exercises a different sequence and a sequence is never exercised more than once [15].

There are several additional ways to reduce the number of sequences exercised during reachability testing:

1. We can use the symmetry of threads to suppress sequences that differ only in the order in which two or more symmetric threads perform the same operation. For example, the three producers in the BB program can enter the monitor in six possible orders, but testing one of these orders is sufficient. Applying this symmetry reduction to the two monitor solutions for the bounded buffer (BB) program reduced the number of exercised sequences from 720 and 12,096 sequences to 20 and 132 sequences, respectively. Likewise, when both the P/V reduction and the symmetry reduction are applied, the number of sequences for the semaphore version of the bounded buffer program (BB-semaphore) is reduced to either three or one (for any number of Producers and Consumers). This is consistent with the analytical results in [16], which showed that the sequences of program BB-semaphore can be "collapsed" into a single representative sequence.
2. In the distributed mutual exclusion (DME) program, processes send requests to each other and wait for replies. However, the behavior of a process does not depend on the values received in the replies or on the identities of the senders or on the order in which replies are received. Instead, it only depends on the number of replies that are received. If we suppress all but one of the possible orders of receiving replies, then the total number of sequences drops from 4,032 to 504.

The above reductions can be performed automatically in RichTest by removing sending events from the race sets of receiving events. The details of how to perform such reductions will be reported in a separate paper.

Another way to speed up reachability testing is to execute multiple instances of RichTest in parallel on multiprocessor/distributed systems. *Distributed reachability testing* partitions the variants among the nodes in a distributed system. Since prefix-based test runs during reachability testing are independent, interprocess communication takes place only when variants are distributed. To get an initial estimate of the speedup that is possible from utilizing multiple processors, we performed an experiment on the dining philosophers (DP) program in Table 4. Program DP had six philosophers and 901,752 possible sequences. It took reachability testing 144.5 minutes to exercise these sequences on a single PC. On a cluster of 10 PCs, distributed reachability testing took only 16.5 minutes. For this experiment, we used a simple partitioning scheme that gave each of the nine "client" nodes an equal number of initial variants and allowed each client node to request more variants from the "server" node when the client ran out. The server could send some of its own variants to a client that requested more or steal variants from some other client. (The server node also performed reachability testing with its own set of variants.) Define the processor utilization factor as the ratio between the execution time of sequential reachability testing and the execution time of distributed reachability testing times the number of clients. The utilization factor for the 10 node experiment was $144.5/(16.5 * 10) \approx 87\%$, which means that the communication overhead used up only 13 percent of the computing power. We believe that we can increase processor utilization by developing specialized load balancing techniques for distributed reachability testing.

9 Related Work

Nondeterministic testing is perhaps the simplest approach to testing concurrent programs. The main problem with this approach is that executions are uncontrolled and, thus, some SYN-sequences may be executed many times, whereas others may never be executed. Techniques have been developed to increase the chances of exercising

different SYN-sequences and, thus, the chances of finding faults when a program is repeatedly executed [10], [27].

A more controlled approach is deterministic testing, which allows carefully selected SYN-sequences to be executed. The main challenge with this approach is how to select a good set of SYN-sequences. There are two general strategies for selecting SYN-sequences. One strategy is to construct an extended control flow graph (CFG) and then select test sequences from the graph [35], [36]. An extended CFG consists of a number of CFGs, one per each process (or thread), that are connected by their static synchronization structure. There are two fundamental problems with this strategy. First, paths selected from an extended CFG may be infeasible, i.e., they cannot be exercised by any program execution. Second, it may be difficult to capture dynamic behaviors (e.g., dynamic thread/process creation, data structures, etc.) in a statically constructed graph.

The alternative strategy for selecting SYN-sequences is to derive the reachability graph of a program (or of a model of the program) and then select test paths from this graph [33], [34]. Every path selected from a reachability graph is guaranteed to be feasible, but this strategy suffers from the state explosion problem, i.e., the number of states in a reachability graph can be enormous for practical applications. Moreover, a reachability graph is fundamentally an interleaving-based concurrency model. Thus, it is possible to select two or more paths that correspond to the same partial order, which is inefficient. The problem of accurately modeling dynamic behaviors also exists in this strategy.

Note that reachability testing combines nondeterministic and deterministic testing. It derives SYN-sequences automatically and on-the-fly, without constructing any static models. Also note that reachability testing is able to systematically exercise every partially ordered SYN-sequence exactly once, which has important applications in program-based verification.

State exploration techniques such as VeriSoft [13] and others [5], [14] have also been developed for testing concurrent programs. These techniques can be considered as model checking applied to programs, instead of their specifications or models. They use partial-order reduction to avoid exercising redundant interleavings of the same partial ordering of events. Partial-order reduction exploits the commutativity of independent transitions and conducts a selective search in which only a subset of the enabled transitions in a global state are explored. Two techniques have been developed for computing these subsets. The first technique is actually a family of algorithms that compute *persistent sets*. This technique needs to identify independent transitions and consider "which operations on which communication objects each thread might execute in the future" [12]. The latter information can be obtained from a static analysis of the program or from the user; otherwise, it must be assumed that anything could happen in the future. The amount of reduction that is obtained depends on the accuracy of this information. The second technique computes subsets of enabled transitions called *sleep sets*. Sleep sets can be used together with persistent sets to further reduce the number of states that are visited. Sleep sets are computed using information in the search history and, if used alone, can reduce the number of transitions that are executed but not the number of states that are visited. VeriSoft uses both persistent set and sleep set techniques for partial order reduction, but no static analysis is performed for computing persistent sets.

We want to point out that VeriSoft explores the state space in a stateless manner, which is accomplished as follows: During state exploration, the sequence of transitions, rather than states, along the current path is stored on the search stack. To restore a state for backtracking, the sequence of transitions, except the last one, on the stack is reexecuted. The history of states that have already been visited is not saved. As a stateless search technique, VeriSoft may visit the same state more than once and can only be used to explore acyclic state spaces. However, being stateless avoids the need to extract an explicit state representation, which is often difficult for programs written in a full-fledged programming language, and the need to store and search the search history, which can be prohibitive for practical applications.

Whereas partial order reduction uses an interleaving-based concurrency model (i.e., a state graph), reachability testing deals with partially ordered SYN-sequences directly. As a result, there is no notion of interleaving in reachability testing and, thus, there is no problem with redundant interleavings. Also, reachability testing performs no static analysis, so it can perform well in cases where partial order reduction does not. This was demonstrated by the results of our case study comparison between RichTest and VeriSoft. Similarly to VeriSoft, reachability testing does not extract or represent any explicit state information nor does it save or search the history of the SYN-sequences that have already been visited.

Dynamic partial order reduction uses dynamic information to compute persistent sets [12]. Dynamic partial order reduction and reachability testing share a similar framework, though they are presented quite differently. Both techniques start with a nondeterministic test run, i.e., an uncontrolled test run that allows nondeterminism to be resolved arbitrarily. At the end of each test run, information that was collected during the run is used to identify branching points and the branching points are used to derive alternative executions that need to be explored. Branching points are called backtracking points in dynamic partial order reduction and race conditions in reachability testing. The alternative executions are then exercised, and new branching points are identified. This procedure is repeated until all the alternative executions are exercised.

Dynamic partial order reduction, like static partial order reduction, is based on an interleaving-based concurrency model and may allow redundant interleavings to be exercised. This is fundamentally different from reachability testing, which represents concurrent executions as partially ordered SYN-sequences and does not have the notion of interleaving. This fundamental difference between the techniques creates differences in the way branching points are identified and alternative executions are derived. We plan to conduct an empirical study that compares reachability testing to dynamic partial order reduction when a dynamic partial order reduction tool becomes available.

Finally, we summarize existing work on reachability testing. In [15], a reachability testing algorithm was described for multithreaded programs that use read and write operations. A reachability testing algorithm for asynchronous message passing programs was reported in [29] and was later improved in [19]. Our work is different in the following aspects: First, we present a general execution model that allows our reachability testing method to be applied to several commonly used synchronization constructs. This is in contrast to existing methods, which are specific to a particular type of synchronization construct. Second, existing reachability testing methods compute race variants by considering all possible interleavings of the events in a SYN-sequence. This is less efficient than our table-based algorithm, which deals with partial orders directly. The notion of a race variant in our work is also slightly different from that in existing methods. Third, in order to guarantee that every partially ordered SYN-sequence is executed exactly once, all the existing reachability testing algorithms need to save the test history, i.e., all the SYN-sequences that have already been exercised, whereas our new reachability testing algorithm saves no test history. We note that the work presented in this paper is an extension of our previous work in [6], [20], [21], [22].

10 Conclusion and Future Work

In this paper, we presented a general execution model, which allows reachability testing to be applied to several commonly used synchronization constructs, and a new method for performing reachability testing. For a closed program whose execution always terminates and whose only source of nondeterminism is due to concurrency, our new method guarantees that every partially ordered SYN-sequence will be exercised exactly once, without saving the history of SYN-sequences that have already been exercised. We consider this new method to be a breakthrough for reachability testing for two reasons. First, removing the need to save and search the test history represents a significant reduction in memory and time requirements and, thus, allows reachability testing to be applied to larger programs. Without the new method, reachability testing simply could not compete with partial order reduction-based search techniques such as VeriSoft. Second, the ability to exercise every partially ordered SYN-sequence exactly once, without any static analysis, is of theoretical interest and has important applications in program-based verification, which is discussed in our future plans. Note that reachability testing tools can be implemented in a portable manner, without modifying the underlying virtual machine, runtime-system, or operating system.

Since reachability testing is implementation-based, it cannot by itself detect "missing sequences," i.e., sequences that are valid according to the specification but are not allowed by the implementation. In this respect, reachability testing is complementary to specification-based testing, which selects valid sequences from a specification and determines whether the sequences are allowed by the implementation. RichTest supports specification-based deterministic testing. Test sequences selected manually or generated from labeled transition systems can be input by RichTest in order to automatically determine whether the sequences are allowed by the implementation [17].

We plan to continue our work in the following directions: First, exhaustive testing is not always practical. In order to enable a balance between test effort and test coverage, we are developing reachability testing algorithms that selectively exercise a set of SYN-sequences according to a specified test coverage criteria. Second, we are addressing the test oracle problem. Reachability testing frequently executes a large number of sequences, which makes it impractical to manually inspect the output of the test executions. At present, properties such as freedom from deadlock and assertion violations can be checked automatically in RichTest. We plan to build new RichTest components that can check advanced temporal properties as well. Third, we will conduct an empirical study to evaluate the fault detection effectiveness of reachability testing. Synchronization faults will be inserted into programs using mutation-based testing techniques. We will determine how many of the inserted faults can be detected by reachability testing. Finally, there is growing interest in combining formal methods and testing. Formal methods are frequently model-based, which means that a model must be extracted from a program. Since reachability testing is dynamic and can be exhaustive, we are investigating the use of reachability testing to construct complete models of the communication and synchronization behavior of concurrent programs.

References

[1] Ada Language Reference Manual, Jan. 1983.

[2] G. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[3] A. Bechini and K.C. Tai, "Timestamps for Programs Using Messages and Shared Variables," *Proc. 18th Int'l Conf. Distributed Computing Systems,* pp. 266-273, 1998.

[4] A. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Trans. Electronic Computers,* vol. 15, no. 5, pp. 757-763, 1966.

[5] D.L. Bruening, "Systematic Testing of Multithreaded Java Programs," master's thesis, Massachusetts Inst. of Technology, 1999.

[6] R. Carver and Y. Lei, "A General Model for Reachability Testing of Concurrent Programs," *Proc. Int'l Conf. Formal Eng. Methods,* pp. 76-98, 2004.

[7] R. Carver and K.C. Tai, "Replay and Testing for Concurrent Programs," *IEEE Software,* vol. 8, no. 2, pp. 66-74, Mar. 1991.

[8] R. Carver and K.C. Tai, "Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs," *IEEE Trans. Software Eng.,* vol. 24, no. 6, pp. 471-490, June 1998.

[9] R. Carver and K.C Tai, *Modern Multithreading*. Wiley, 2005.

[10] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithread Java Program Test Generation," *IBM Systems J.,* vol. 41, no. 1, pp. 111-125, 2002.

[11] C.J. Fidge, "Logical Time in Distributed Computing Systems," *Computer,* pp. 28-33, Aug. 1991.

[12] C. Flanagan and P. Godefroid, "Dynamic Partial Order Reduction for Model Checking Software," *Proc. 32nd Symp. Principles of Programming Languages (POPL),* pp. 110-121, 2005.

[13] P. Godefroid, "Software Model Checking: The VeriSoft Approach," *Formal Methods in System Design,* vol. 26, no. 2, pp. 77-101, 2005.

[14] K. Havelund and T. Pressburger, "Model Checking Java Programs Using Java PathFinder," *Int'l J. Software Tools for Technology Transfer (STTT),* vol. 2, no. 4, pp. 366-381, Apr. 2000.

[15] G.H. Hwang, K.C. Tai, and T.L. Huang, "Reachability Testing: An Approach to Testing Concurrent Software," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 5, no. 4, pp. 493-510, 1995.

[16] S. Katz and D. Peled, "Defining Conditional Independence Using Collapses," *Theoretical Computer Science,* vol. 101, pp. 337-359, 1992.

[17] P.V. Koppol, R.H. Carver, and K.C. Tai, "Incremental Integration Testing of Concurrent Programs," *IEEE Trans. Software Eng.*, vol. 28, no. 6, pp. 607-623, June 2002.
[18] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, pp. 558-565, July 1978.
[19] Y. Lei and K.C. Tai, "Efficient Reachability Testing of Asynchronous Message-Passing Programs," *Proc. Eighth IEEE Int'l Conf. Eng. for Complex Computer Systems*, pp. 35-44, Dec. 2002.
[20] Y. Lei and R.H. Carver, "Reachability Testing of Semaphore-Based Programs," *Proc. 28th Computer Software and Applications Conf. (COMPSAC)*, pp. 312-317, 2004.
[21] Y. Lei and R. Carver, "Reachability Testing of Monitor-Based Programs," *Proc. Int'l Conf. Software Eng. and Applications*, pp. 312-317, 2004.
[22] Y. Lei and R. Carver, "A New Algorithm for Reachability Testing of Concurrent Programs," *Proc. Int'l Symp. Software Reliability Eng.*, pp. 346-355, 2005.
[23] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, M. Cosnard et al., eds., pp. 215-226, North Holland: Elsevier Science, 1989.
[24] R.H.B. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," *Proc. Third ACM/ONR Workshop Parallel and Distributed Debugging*, pp. 1-11, 1993.
[25] G. Ricart and A.K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Comm. ACM*, vol. 24, no. 1, pp. 9-17, Jan. 1981.
[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Race Detector for Multithreaded Programs," *IEEE Trans. Computer Systems*, vol. 15, no. 4, pp. 391-411, 1998.
[27] S.D. Stoller, "Testing Concurrent Java Programs Using Randomized Scheduling," *Proc. Second Workshop Runtime Verification (RV)*, 2002.
[28] K.C. Tai, "Race Analysis of Traces of Asynchronous Message-Passing Programs," *Proc. 17th Int'l Conf. Distributed Computing Systems*, pp. 261-268, 1997.
[29] K.C. Tai, "Reachability Testing of Asynchronous Message-Passing Programs," *Proc. Second Int'l Workshop Software Eng. for Parallel and Distributed Systems*, pp. 50-61, 1997.
[30] K.C. Tai, R.H. Carver, and E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution," *IEEE Trans. Software Eng.*, vol. 17, no. 1, pp. 45-63, Jan. 1991.
[31] K.C. Tai and R.H. Carver, "Testing of Distributed Programs," *Handbook of Parallel and Distributed Computing*, A. Zoyama, ed., chapter 33, pp. 955-978, McGraw-Hill, 1996.
[32] R.N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs," *Comm. ACM*, vol. 26, no. 5, pp. 362-376, 1983.
[33] R.N. Taylor, D.L. Levine, and C.D. Kelly, "Structural Testing of Concurrent Programs," *IEEE Trans. Software Eng.*, vol. 18, no. 3, pp. 206-214, Mar. 1992.
[34] A. Ulrich and H. Konig, "Specification-Based Testing of Concurrent Systems," *Proc. IFIP Joint Int'l Conf. Formal Description Techniques and Protocol Specification, Testing, and Verification (FORTE/PSTV '97)*, pp. 7-22, 1997.
[35] R.D. Yang and C.G. Chung, "A Path Analysis Approach to Concurrent Program Testing," *Information and Software Technology*, vol. 34, no. 1, pp. 43-56, 1992.
[36] C. Yang, A.L. Souter, and L.L. Pollock, "All-du-Path Coverage for Parallel Programs," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*, pp. 153-162, 1998.

Yu Lei received the BS degree in computer science from Wuhan University in 1993, the MS degree in software engineering in 1996 from the Institute of Software, Chinese Academy of Sciences, and the PhD degree in computer science in 2002 from North Carolina State University. He is currently an assistant professor of computer science at the University of Texas at Arlington. His current research interests are in the area of automated software analysis, testing, and verification. He is a member of the IEEE and the IEEE Computer Society.

Richard H. Carver received the BS degree in computer science from the Ohio State University in 1982, the MS degree in computer studies in 1985, and the PhD degree in computer engineering in 1989, both from North Carolina State University. He is currently an associate professor of computer science at George Mason University. His current research interests include the analysis, testing, and debugging of concurrent software.