A General Model for Reachability Testing of Concurrent Programs

Richard H. Carver[1] and Yu Lei[2]

[1] Dept. of Computer Science, MS 4A5,
George Mason University, Fairfax
VA 22030-4444
rcarver@cs.gmu.edu
[2] Dept. of Computer Science and Engineering,
The University of Texas at Arlington, Arlington, Texas, 76019
ylei@cse.uta.edu

Abstract. Reachability testing is a technique for testing concurrent programs. Reachability testing derives test sequences on-the-fly as the testing process progresses, and can be used to systematically exercise all the behaviors of a program. The main contribution of this paper is a general model for reachability testing. This model allows reachability testing to be applied to many different types of concurrent programs, including asynchronous and synchronous message passing programs, and shared-memory programs that use semaphores, locks, and monitors. We define a common format for execution traces and present timestamp assignment schemes for identifying races and computing race variants, which are a crucial part of reachability testing. Finally, we discuss a prototype reachability testing tool, called RichTest, and present some empirical results.

1 Introduction

Concurrent programming is an important technique in modern software development. Concurrency can improve computational efficiency and resource utilization. However, concurrent programs behave differently than sequential programs. Multiple executions of a concurrent program with the *same* input may exercise *different* sequences of synchronization events (or SYN-sequences) and produce *different* results. This non-deterministic behavior makes concurrent programs notoriously difficult to test.

A simple approach to dealing with non-deterministic behavior when testing a concurrent program CP is to execute CP with a fixed input many times and hope that faults will be exposed by one of these executions [18]. This type of testing, called *non-deterministic testing*, is easy to carry out, but it can be very inefficient. It is possible that some behaviors of CP are exercised many times while others are never exercised. An alternative approach is called *deterministic testing*, which forces a specified SYN-sequence to be exercised. This approach allows CP to be tested with carefully selected SYN-sequences. The test sequences are usually selected from a

J. Davies et al. (Eds.): ICFEM 2004, LNCS 3308, pp. 76–98, 2004.
Springer-Verlag Berlin Heidelberg 2004

static model of CP or of CP's design. However, accurate static models are often difficult to build for dynamic behaviors.

Reachability testing is an approach that combines non-deterministic and deterministic testing [9] [12] [19]. It is based on a technique called prefix-based testing, which controls a test run up to a certain point, and then lets the run continue non-deterministically. The controlled portion of the execution is used to force the execution of a "prefix SYN-sequence", which is the beginning part of one or more feasible SYN-sequences of the program. The non-deterministic portion of the execution exercises one of these feasible sequences.

A novel aspect of reachability testing is that it adopts a dynamic framework in which test sequences are derived automatically and on-the-fly, as the testing process progresses. In this framework, synchronization events (or SYN-events) are recorded in an execution trace during each test run. At the end of a test run, the trace is analyzed to derive prefix SYN-sequences that are "race variants" of the trace. A race variant represents the beginning part of a SYN-sequence that definitely could have happened but didn't, due to the way race conditions were arbitrarily resolved during execution. The race variants are used to conduct more test runs, which are traced and then analyzed to derive more race variants, and so on. If every execution of a program with a given input terminates, and the total number of SYN-sequences is finite, then reachability testing will terminate and every partially-ordered SYN-sequence of the program with the input will be exercised.

Reachability testing requires program executions to be modeled so that races can be identified and race variants can be generated. The execution model must also contain sufficient information to support execution tracing and replay. Models for tracing and replay have been developed for many synchronization constructs, including semaphores, locks, monitors, and message passing [4] [20]. However, these models do not support race analysis. Models for race analysis have been developed for message passing, but not for other synchronization constructs. The contributions of this paper are: (1) a general execution model for reachability testing that supports race analysis and replay for all of the synchronization constructs mentioned above. This model defines a common format for execution traces and provides a timestamp assignment scheme that assists in identifying races and computing race variants. (2) A race analysis method that can be used to identify races in executions captured by our execution model. This method can be used by an existing algorithm for generating race variants. (3) A Java reachability testing tool, called RichTest, that implements reachability testing without any modifications to the Java JVM or to the operating system.

The rest of this paper is organized as follows. The next section illustrates the reachability testing process. Section 3 presents an execution model for several commonly used synchronization constructs. Section 4 defines the notions of a race and a race variant, and discusses how to identify races and compute race variants. Section 5 describes the RichTest tool and reports some empirical results. Section 6 briefly surveys related work. Section 7 provides concluding remarks and describes our plans for future work.

2 The Reachability Testing Process

We use a simple example to illustrate the reachability testing process. Fig. 1 shows a program CP that consists of four threads. The threads synchronize and communicate by sending messages to, and receiving messages from, ports. Ports are communication objects that can be accessed by many senders but only one receiver. Each send operation specifies a port as its destination, and each receive operation specifies a port as its source.

Fig. 1 also shows one possible scenario for applying reachability testing to the example program. Each sequence and race variant generated during reachability testing is represented by a space-time diagram in which a vertical line represents a thread, and a single-headed arrow represents asynchronous message passing between a send and receive event. The labels on the arrows match the labels on the send and receive statements in program CP. The reachability testing process in Fig. 1 proceeds as follows:

- First, sequence Q0 is recorded during a non-deterministic execution of CP. Sequence V1 is a race variant of Q0 derived by changing the outcome of a race condition in Q0. That is, in variant V1, thread T3 receives its first message from T4 instead of T2. The message sent by T2 is left un-received in V1.
- During the next execution of CP, variant V1 is used for prefix-based testing. This means that variant V1 is replayed and afterwards the execution proceeds non-deterministically. Sequence Q1 is recorded during this execution. Sequence Q1 is guaranteed to be different from Q0 since V1 and Q0 differ on the outcome of a race condition and V1 is a prefix of Q1. Variant V2 is a race variant of Q1 in which T2 receives its first message from T3 instead of T1.
- When variant V2 is used for prefix-based testing, sequence Q2 is recorded. Reachability testing stops since Q0, Q1 and Q2 are all the possible SYN-sequences that can be exercised by this program.

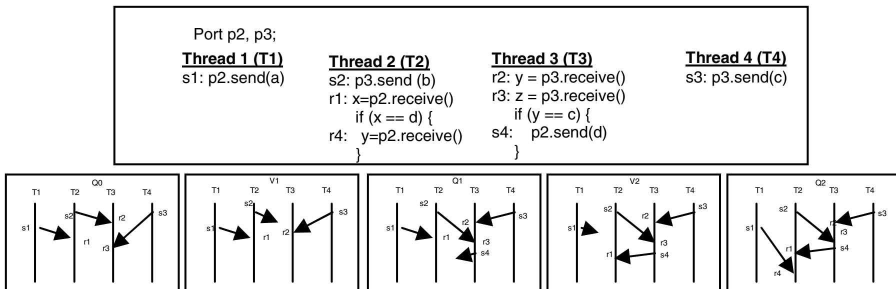


Fig. 1. The reachability testing process

For a formal description of the above process, the reader is referred to a reachability testing algorithm that we reported in [13]. The challenge for reachability

testing is to identify races and derive race variants. This is discussed in the remainder of this paper.

3 Models of Program Executions

In this section, we present a general execution model for several commonly used synchronization constructs. This model provides sufficient information for replaying an execution and for identifying the race variants of an execution. Replay techniques have already been developed for these constructs [4] [20]. An algorithm for computing race variants is described in the next section.

3.1 Asynchronous Message Passing

Asynchronous message passing refers to non-blocking send operations and blocking receive operations. A thread that executes a non-blocking 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. We assume that asynchronous ports (see Section 2) have unlimited capacity (which means that a send operation is never blocked) and use a FIFO (First-In-First-Out) message ordering scheme, which guarantees that messages passed between any two threads are received in the order that they are sent.

```
Port p;
Thread 1       Thread 2
p.send(msg)    msg = p.receive();
```

An execution of a program that uses asynchronous ports exercises a sequence of send and receive events. A send or receive event refers to the execution of a send or receive statement, respectively. A send event s and the receive event *r* it synchronizes with forms a synchronization pair $<s, r>$, where *s* is said to be the send partner of *r*, and *r* is said to be the receive partner of s. We use an event descriptor to encode certain information about each event. Each send event *s* is assigned an event descriptor (T, O, i), where *T* is the sending thread, *O* is the port, and *i* is the event index indicating that *s* is the *i*-th event in *T*. Each receive event *r* is assigned an event descriptor (T, O, i), where *T* is the receiving thread, *O* is the port name, and *i* is the event index indicating that r is the *i*-th event of *T*. A send event *s* is said to be open at a receive event *r* if $s.O = r.O$.

Fig. 2 shows a space-time diagram representing an execution with three threads. Thread *T2* receives messages from ports *p1* and *p2*. Thread *T1* sends two messages to port *p1*. Thread *T3* sends its first message to port *p1* and its second message to port *p2*.

We note that in many applications a thread only has one port for receiving messages. In this special case, a thread identifier is usually specified as the destination of a send event, and the source of a receive event can be left unspecified. Also, a link-based communication scheme can be simulated by using ports that are restricted to having only one sender. We also point out that in practical implementations, ports are often implemented using bounded buffers that can only hold a fixed number of

messages. In this case, a send operation can be blocked if the capacity of a buffer is reached. Our model can be applied to buffer-blocking ports without any modification.

3.2 Synchronous Message Passing

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

A selective wait construct is commonly used in synchronous message passing to allow a combination of waiting for, and selecting from, one or more receive() alternatives [1]. The selection can depend on guard conditions associated with each alternative of the selective wait:

```
Port port1, port2;
select
  when (guard condition 1) => port1.receive();
or
  when (guard condition 2) => port2.receive();
end select;
```

A receive alternative is said to be *open* if it does not start with when(guard condition), or if the value of the guard condition is *true*. It is said to be *closed* otherwise. A select statement works as follows:

- an open receive-alternative (i.e., one with a *true* guard) is selected only if that alternative has a waiting message.
- if several receive-alternatives are open and have waiting messages, the alternative whose message arrived first is selected.
- if one or more receive-alternatives are open but none have a waiting message, select blocks until a message arrives for one of the open receive-alternatives.
- If none of the receive-alternatives are open, select throws an exception.

We make the restriction that there can be only one receive-alternative for a given port.

A send event s and the receive event r it synchronizes with form a rendezvous pair $<s, r>$, where s is the send partner of r and r is the receive partner of s. Each send event s is assigned an event descriptor (T, O, i), where T is the sending thread, O is the port, and i is the event index indicating that s is the i-th event of T. Each receive event r is assigned an event descriptor (T, L, i), where T is the receiving thread, L is the open-list of r, and i is the index indicating that r is the i-th event of T. The open-list of a receive event r is a list containing the ports that had open receive-alternatives at r. Note that this list includes the source port of r. For a simple receive statement that is not in a selective wait, the list of open alternatives consists of the source port of the receive statement only. Event s is said to be open at r if the port $s.O$ of s is in the open-list $r.L$ of r.

Fig. 3 shows a space-time diagram representing an execution with three threads. 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*. Assume that whenever *p2* is selected, the alternative for *p1* is open, and whenever *p1* is selected, the alternative for *p2* is closed. This is reflected in the open-lists for the receive events. Note that each solid arrow is followed by a dashed arrow in the opposite direction. The dashed arrows represent the updating of timestamps when the synchronous communication completes, and will be discussed in Section 4.

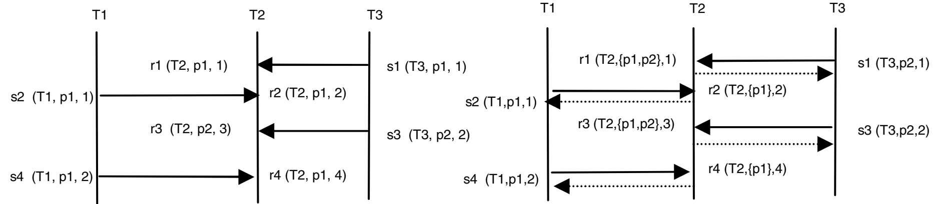


Fig. 2. A sequence of asynchronous send/ receive events

Fig. 3. A sequence of synchronous send/ receive events

3.3 Semaphores

A semaphore is a synchronization object that is initialized with an integer value and then accessed through two operations named *P* and *V*. Semaphores are provided in many commercial operating systems and thread libraries. There are two types of semaphores – counting semaphores and binary semaphores.

A *V()* operation on a counting semaphore *s* increments the value of *s*. A *P()* operation decrements the value of *s*, but if *s* is less than or equal to zero when the *P()* operation starts, the *P()* operation waits until *s* is positive. For a *counting semaphore s*, at any time, the following relation, called the semaphore invariant, holds:

(initial value of *s*) + (number of completed *s.V()* operations) ≥ (number of completed *s.P()* operations)

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.

A *binary semaphore* must be initialized with the value 1 or the value 0 and the completion of *P()* and *V()* operations must alternate. (*P()* and *V()* operations can be started in any order, but their completions must alternate.) If the initial value of the semaphore is 1 the first completed operation must be *P()*. If a *V()* operation is attempted first, the *V()* operation will block its caller. Likewise, if the initial value of the semaphore is 0, the first completed operation must be *V()*. Thus, the *P()* and *V()* operations of a binary semaphore may block the calling threads. (Note that *V()* operations are sometimes defined to be non-blocking – executing a non-blocking *V()* operation on a binary semaphore has no effect if the value of the semaphore is 1. In this paper, we are using a blocking *V()* operation. Our model can be easily adjusted if a non-blocking *V()* operation is used.) We assume that the queues of blocked threads are FIFO queues.

We model the invocation of a *P()* or *V()* operation as a pair of call and completion events. When a thread *T* calls a *P()* or *V()* operation on a semaphore *S*, a "semaphore-call" event, or simply a "call" event, *c* is performed by *T*. When a *P()* or *V()* operation of a semaphore *S* is completed, a "semaphore-completion" event, or simply a "completion" event, *e* occurs on *S*. If the operation of a call event *c* is completed by a completion event *e*, we say that *c* and *e* form a completion pair <*c, e*>, where *c* is the call partner of *e* and *e* is the completion partner of *c*. This model is intentionally similar to the model for message passing where a synchronization pair was defined as a pair of send and receive events.

Each call event *c* is assigned a descriptor *(T, S, op, i)*, where *T* is the calling thread, *S* is the destination semaphore, *op* is the called operation (*P()* or *V()*), and *i* is the event index indicating that *c* is the *i*-th (call) event performed by *T*. A completion event *e* is assigned a descriptor *(S, L, i)*, where *S* is the semaphore on which *e* occurs, *L* is the list of operations (*P()* and/or *V()*) that can be completed at *e*, and *i* is the event index indicating that *e* is the *i*-th (completion) event that occurs on *S*. *L* is also called the open-list of *e*. A call event *c* is open at a completion event *e* if *c.S* = *e.S*, and the operation *c.op* of *c* is in the open-list *e.L* of *e*.

Fig. 4 shows a space-time diagram representing an execution with two threads *T1* and *T2*, and a binary semaphore *S* initialized to 1. Each of *T1* and *T2* performs a *P()* and *V()* operation on *S*. In this diagram, semaphore *S* is also represented as a vertical line, which contains the entry events that occurred on *S*. A solid arrow represents the completion of a *P()* or *V()* operation. The open-lists for the completion events model the fact that P and V operations on a binary semaphore must alternate. Note that each solid arrow is followed by a dashed arrow in the opposite direction. The dashed arrows represent the updating of timestamps when operations complete, and will be discussed in Section 4.

3.4 Locks

A mutex (for "*mut*ual *ex*clusion") lock is a synchronization object that is used to create critical sections. The operations on a mutex lock are named *lock*() and *unlock*(). Unlike semaphores, a mutex lock has an owner, and ownership plays an important role in the behavior of a mutex lock:

- A thread requests ownership of mutex lock *K* by calling *K.lock().*
- A thread that calls *K.lock()* becomes the owner if no other thread owns the lock; otherwise, the thread is blocked.
- A thread releases its ownership of *K* by calling *K.unlock().* If the thread does not own *K*, the call to *K.unlock()* generates an error.
- A thread that already owns lock *K* and calls *K.lock()* again is not blocked. In fact, it is common for a thread to request and receive ownership of a lock that it already owns. But the thread must call *K.unlock()* the same number of times that it called *K.lock(),* before another thread can become *K*'s owner.

Our model for *lock()* and *unlock()* operations on mutex locks is similar to our model for *P()* and *V()* operations on semaphores. When a thread *T* calls a *lock()* or *unlock()* operation on mutex lock *K*, a "mutex-call" event, or simply a "call" event, *c*

occurs on *T*. When *T* eventually finishes a *lock()* or *unlock()* operation, a "mutex-completion" event, or simply a "completion" event, *e* occurs on *K*. If the operation of a call event *c* is completed by a completion event *e*, we say that *c* and *e* form a completion pair <*c,e*>, where *c* is the call partner of *e* and *e* is the completion partner of *c*.

Each call event *c* is assigned a descriptor *(T, K, op, i)*, where *T* is the calling thread, *K is* the destination lock, *op* is the called operation (*lock()* or *unlock()*), and *i* is the event index indicating that *c* is the *i*-th (call) event performed by *T*. A completion event *e* is assigned a descriptor *(K, L, i)*, where *K* is the lock on which *e* occurs, *L* is the list of operations (*lock()* and/or *unlock()*) that can be completed at *e*, and *i* is the event index indicating that *e* is the *i*-th (completion) event that occurs on *K*. *L* is also called the open-list of *e*. If the lock is owned by some thread *T* when *e* occurs, then each operation in *L* is prefixed with *T* to indicate that only *T* can perform the operation. This is because if a thread *T* owns lock *L*, then only *T* can complete a *lock()* or *unlock()* operation on *L*. For example, if the open-list *L* of an entry event *e* on a lock *K* contains two operations *lock()* and *unlock()*, and if *K* is owned by a thread *T* when *e* occurs, then *L = {T:lock(), T:unlock()}*. A call event *c* executed by thread *T* is open at a completion event *e* if *c.K = e.K*, and the operation *c.op* of *c* is in the open-list *e.L* of *e*, and if *K* is owned when *e* occurs then *T* is the owner.

Fig. 5 shows a space-time diagram representing an execution with two threads and a mutex lock *K* initialized to 1. Thread *T1* performs two *lock()* operations followed by two *unlock()* operations on *K*, and thread *T2* performs one *lock()* operation followed by one *unlock()* operation on *K*.

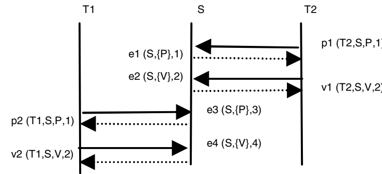


Fig. 4. A sequence of P and V events

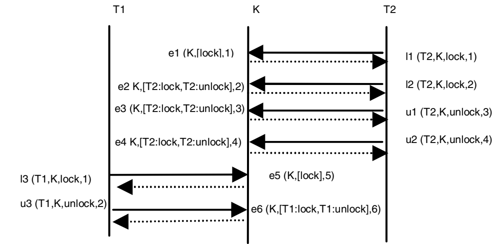


Fig. 5. A sequence of lock and unlock events

3.5 Monitors

A monitor is a high-level synchronization construct that supports data encapsulation and information hiding and is easily adapted to an object oriented environment. We use an object oriented definition of a monitor in which a monitor is a synchronization object that is an instance of a special "monitor class". The data members of a monitor represent shared data. Threads communicate by calling monitor methods that access the shared data.

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. If a thread calls a monitor method, but another

thread is already executing inside the monitor, the calling thread must wait outside the monitor. A monitor has an *entry* queue to hold the calling threads that are waiting to enter the monitor.

Condition 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. When the *Signal-and-Continue* (*SC*) discipline is used, the signaling thread continues to execute in the monitor, and the signaled thread does not reenter the monitor immediately. We assume that the signaled thread joins the entry queue and thus competes with calling threads to enter the monitor. When the *Signal-and-Urgent-Wait* (*SU*) discipline is used, the signaling thread is blocked in a queue called the *reentry* queue and the signaled thread reenters the monitor immediately. The difference between the *entry* and *reentry* queues is that the former holds calling threads that are waiting to enter the monitor for the first time while the latter holds threads that have entered the monitor, executed a *signal* operation, and are waiting to reenter the monitor. The *SU* discipline assigns a higher priority to the reentry queue, in the sense that a thread in the entry queue can enter the monitor only if the reentry queue is empty.

We assume that a monitor's entry queue and the queues associated with condition variables are FIFO queues. Thus, the only non-determinism that is present in a monitor is the order in which threads (re)enter the monitor. Such monitors enjoy a beneficial property called entry-based execution, i.e., the execution behavior of threads inside a monitor is completely determined by the order in which the threads (re)enter the monitor and the values of the parameters on the calls to the monitor methods [4]. Therefore, an entry-based execution can be replayed by replaying the sequence of (re)entry events, called the Entry-sequence, exercised by this execution. Note that an entry event is an event that occurs when a thread enters an *SU* or *SC* monitor for the *first* time or when a thread reenters an *SC* monitor after being signaled. Reentries into an *SU* monitor are not modeled because they do not involve any races. A replay technique for monitor-based programs with entry-based executions was described in [4]. In the remainder of this paper, we assume that monitor-based programs have entry-based executions and the order of the entries is the sole source of non-determinism in the programs.

Characterizing a monitor-based execution as an Entry-sequence is sufficient for replaying executions, but not for identifying races. When two or more threads call a monitor at the same time, they race to see which one will enter first. Thus, we model the invocation of a monitor method as a pair of monitor-call and monitor-entry events:

- *SU Monitors*: When a thread T calls a method of monitor M, a monitor-call event, or simply a call event, c occurs on T. When T eventually enters M, a monitor-entry event, or simply an entry event, e occurs on M, and then T starts to execute inside M.
- *SC Monitors*: When a thread T calls a method of monitor M, a monitor-call event, or simply a call event, c occurs on T. A call event also occurs when T tries to reenter a monitor M after being signaled. When T eventually (re)enters M, a

monitor-entry event, or simply an entry event, *e* occurs on *M*, and *T* starts to execute inside *M*.

In these scenarios, we say that *T* is the calling thread of *c* and *e*, and *M* is the destination monitor of *c* as well as the owning monitor of *e*. We also say that *c* and *e* form an entry pair <*c, e*>, where *c* is the call partner of *e* and *e* is the entry partner of *c*.

Each call event *c* is assigned an event descriptor *(T, M, i)*, where *T* is the calling thread, *M* is the destination monitor, and *i* is the event index indicating that *c* is the *i*-th (call) event of *T*. Each entry event *e* is assigned an event descriptor *(M, i)*, where *M* is the owning monitor, and *i* is the event index indicating that *e* is the *i*-th event of *M*. A call event *c* is open at an entry event *e* if the destination monitor of *c* is the owning monitor of *e*, i.e., *c.M = e.M*.

Fig. 6 shows a space-time diagram, which represents an execution involving three threads *T1*, *T2*, and *T3*, and two *SC* monitors *M1* and *M2*. Thread *T1* enters *M1* first and executes a *wait()* operation. The second call event performed by *T1* occurs when *T1* reenters *M1* after being signaled by *T2*. Note that if *M1* were an *SU* monitor, there would be no *c3* event representing reentry. After *T1* exits from *M1*, *T1* enters and exits *M2*. This is followed by thread *T3* entering and exiting *M2* and then entering and exiting *M1*.

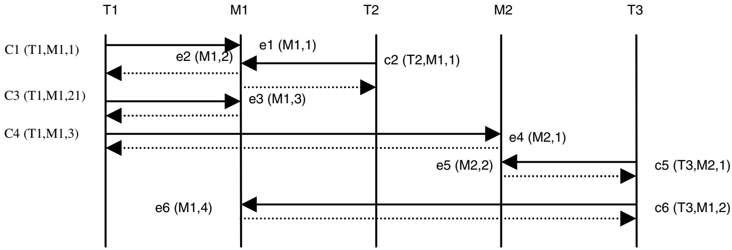


Fig. 6. A sequence of monitor call and entry events

3.6 A General Model

In the models presented above, a program execution is characterized as a sequence of event pairs. For asynchronous and synchronous message-passing programs, an execution is characterized as a sequence of send and receive events. For semaphore-, lock-, and monitor-based programs, an execution is characterized as a sequence of call and completion events. In the remainder of this paper, we will refer to a send/call event as a sending event, and a receive/completion event as a receiving event. We also refer to a pair of sending and receiving events as a synchronization pair.

The event descriptors for the sending and receiving events defined above all fit into a single general format:

- A descriptor for a sending event *s* is denoted by *(T, O, op, i)*, where *T* is the thread executing the sending event, *O* is the destination object, *op* is the operation performed, and *i* is the event index indicating that *s* is the *i*-th event of *T*. Note that

for message passing, *op* is always a send operation, and for monitors, *op* is the called method.

- A descriptor for a receiving event *r* is denoted by *(D, L, i)*, where *D* is the destination thread or object, *L* is the open-list, and *i* is the event index indicating that *r* is the *i*-th event of *D*. Note that for asynchronous message-passing, *L* contains the source port of *r* only, and is thus represented as a single port. For a monitor, *L* contains all of the methods defined on the monitor since entry into a monitor is never guarded. (A thread may be blocked after it enters a monitor, but a thread that calls a monitor method is guaranteed to eventually enter the method.)

In programs that use shared variables, we assume that accesses to shared variables are always protected by semaphores, locks, or monitors. To enforce this, reachability testing can be used in conjunction with the techniques used in data race detection tools for multithreaded programs [16].

4 Race Analysis of SYN-Sequences

In this section, we show how to perform race analysis on SYN-sequences. Section 4.1 presents two schemes for assigning logical timestamps to determine the happened-before relation between events. Section 4.2 defines the notion of a race and shows how to identify races based on the happened-before relation. Section 4.3 defines the notion of a race variant and uses an example to illustrate how to compute race variants.

4.1 Timestamp Assignment

As we will see in Section 4.2, the definition of a race between events in a SYN-sequence is based on the happened-before relation, which is a partial order defined in the traditional sense [11]. Simply put, an event *a* happens before another event *b* in a *SYN*-sequence *Q* if *a* could potentially affect *b*. We denote this as $a \rightarrow_Q b$, or simply $a \rightarrow b$ if *Q* is implied. In a space-time diagram, if we take into account the direction of the (solid and dashed) arrows, *a* happens before *b* if there exists a path from *a* to *b*.

Vector timestamps are frequently used to capture the happened-before relation between events. In this section, we present thread-centric and object-centric timestamp assignment schemes. A *thread-centric* timestamp has a dimension equal to the number of threads involved in an execution. An *object-centric* timestamp has a dimension equal to the number of synchronization objects involved. Therefore, a thread-centric scheme is preferred when the number of threads is smaller than the number of synchronization objects, and an object-centric scheme is preferred otherwise. In the remainder of this section, we will use *v[i]* to denote the *i*-th component of a vector *v*, and $max(v_1, v_2)$ to denote the component-wise maximum of vectors v_1 and v_2.

4.1.1 A Thread-Centric Scheme

A vector timestamp scheme for asynchronous message passing programs has already been developed [6][14]. This scheme is thread-centric by our definition and can be

used for race analysis. In this scheme, each thread maintains a vector clock. A vector clock is a vector of integers 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 copy of the vector clock as its timestamp.

Let *T.v* be the vector clock maintained by a thread *T*. Let *f.ts* be the vector timestamp of an event *f*. 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 in asynchronous message passing programs:

1. When a thread T_i executes a non-blocking send event *s*, it performs the following operations: (a) $T_i.v[i] = T_i.v[i] + 1$; (b) $s.ts = T_i.v$. Thread T_i also sends *s.ts* along with the message sent by *s*.
2. When a thread T_j executes a receive event *r*, it performs the following operations: (a) $T_j.v[j] = T_j.v[j] + 1$; (b) $T_j.v = max(T_j.v, s.ts)$; (c) $r.ts = T_j.v$, where *s* is the synchronization partner of *r*.

Fig. 7(a) shows the timestamps for the asynchronous message passing program in Fig. 2.

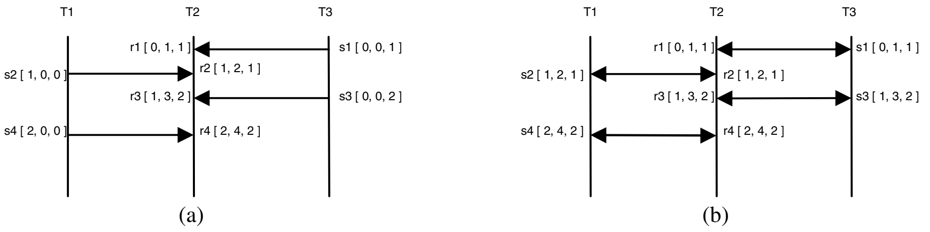


Fig. 7. Traditional timestamp schemes for asynchronous and synchronous message passing

A timestamp scheme for synchronous message passing has also been developed [6], but this scheme must be extended for race analysis. The traditional timestamp scheme for synchronous message passing is to assign the same timestamp to send and receive events that are synchronization partners:

1. When a thread T_i executes a blocking send event *s*, it performs the operation $T_i.v[i] = T_i.v[i] + 1$. Thread T_i also sends $T_i.v$ along with the message sent by *s*.
2. When a thread T_j executes a receiving 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) $T_j.v = max(T_j.v, T_i.v)$; (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) $T_i.v = max(T_i.v, T_j.v)$; (b) $s.ts = T_i.v$.

The exchange of vector clock values between threads T_i and T_j represents the synchronization that occurs between them, which causes their send and receive events to be completed at the same time. Fig. 7b shows the timestamps for the synchronous message passing program in Fig. 3.

In our execution model for synchronous message passing, we model the start of a send event, not its completion. For send and receive events that are synchronization partners, the start of the send is considered to happen before the receive event with which the send eventually synchronizes. This means that the timestamps for send and receive partners should not be the same. Thus, we use the timestamp of the receive event to update the vector clock of the sending thread, but not the timestamp of the send event, when the synchronization completes:

1. When a thread T_i executes a blocking send event s, it performs the following operations: (a) $T_i.v[i] = T_i.v[i] + 1$. (b) $s.ts = T_i.v$. Thread T_i also sends $T_i.v$ along with the message sent by s.
2. When a thread T_j executes a receiving 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) $T_j.v = max(T_j.v, T_i.v)$; (b) $r.ts = T_j.v$. Thread T_j also sends $T_j.v$ back to thread T_i.
3. Thread T_i receives $T_j.v$ and performs the operation (a) $T_i.v = max(T_i.v, T_j.v)$.

Fig. 8 shows the timestamps that are assigned so that race analysis can be performed on the synchronous message passing program in Fig. 3. Note that the dashed arrows represent the execution of rule 3.

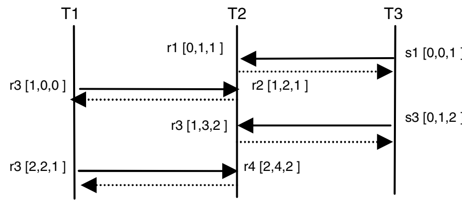


Fig. 8. Timestamp scheme for race analysis of synchronous message passing programs

Below we describe a new thread-centric timestamp scheme for semaphores, locks, and monitors. We refer to semaphores, locks, and monitors generally as "synchronization objects". 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 do not have integer clocks and thus there are no positions in a vector clock for the synchronization objects.) Let $T.v$ (or $O.v$) be the vector clock maintained by a thread T (or a 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) $T_i.v[i] = T_i.v[i] + 1$; (b) $s.ts = T_i.v$;
2. When a receiving event r occurs on a synchronization object O, the following operations are performed: (a) $O.v = max(O.v, s.ts)$; (b) $r.ts = O.v$, where s is the sending partner of r;

3. Semaphore/Lock: When a thread T_i finishes executing an operation on a semaphore or lock O, it updates its vector clock using the component-wise 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 on a monitor O, it updates its vector clock using the component-wise maximum of $T.v$ and $O.v$, i.e., $T.v = max(T.v, O.v)$.

   SC Monitor: When a thread T_i finishes executing a method on a monitor O, or when a thread T_i is signaled from a condition queue of O, it updates its vector clock using the component-wise maximum of $T_i.v$ and $O.v$, i.e., $T_i.v = max(T_i.v, O.v)$.

Figs. 9a and 9b shows the thread-centric timestamps assigned for the executions in Figs. 4 and 6, respectively. Again, dashed arrows are also shown to indicate applications of the third rule.

Thread-centric timestamps can be used to determine the happened-before relation between two arbitrary events, as shown below.

Proposition 1. Let *X* be an execution involving threads $T_1, T_2, \ldots, T_n$ and semaphores, locks, or monitors. Let *Q* be the *SYN*-sequence exercised by *X*. Assume that every event in *Q* is assigned a thread-centric timestamp. Let *f.tid* be the (integer) thread ID of thread *f.T* for an event *f*. Let f_1 and f_2 be two events in *Q*. Then, $f_1 \rightarrow f_2$ if and only if

(1) $<f_1, f_2>$ is a synchronization pair; or
(2) $f_1.ts[f_1.tid] \leq f_2.ts[f_1.tid]$ and $f_1.ts[f_2.tid] < f_2.ts[f_2.tid]$.

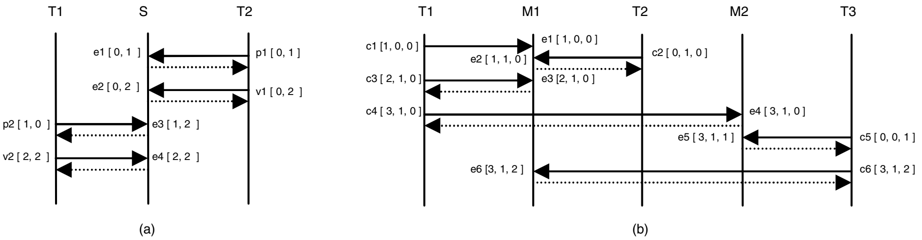


Fig. 9. Timestamp scheme for race analysis of semaphore-based and monitor-based programs

4.1.2 Object-Centric Scheme

In this scheme, each thread and synchronization object (port, semaphore, lock, or monitor) maintains a version vector. A version vector is a vector of integers used to keep track of the version number of each synchronization object. The version number of a synchronization object is initially zero, and is incremented each time a thread performs a sending or receiving event. Each sending and receiving event is also assigned a version vector as its timestamp.

Let *T.v* (or *O.v*) be the version vector maintained by a thread *T* (or a synchronization object *O*). Initially, the version vector of each thread or

synchronization object is a vector of zeros. The following rules are used to update version vectors and assign timestamps to events:

1. When a thread T executes a sending event s, T assigns its version vector as the timestamp of s, i.e., $s.ts = T.v$;.
2. When a receiving event r occurs on a synchronization object O_i, letting s be the sending partner of r, the following operations are performed: (a) $O_i.v = max(O_i.v, s.ts)$; (b) $r.ts = O_i.v$.
3. Semaphore/Lock: When a thread T finishes a called operation on a semaphore or lock O, T updates its version vector using the component-wise maximum of $T.v$ and $O.v$, i.e., $T.v = max(T.v, O.v)$.
   SU Monitor: When a thread T finishes executing a method on a monitor O, T updates its version vector using the component-wise maximum of $T.v$ and $O.v$, i.e., $T.v = max(T.v, O.v)$.
   SC Monitor: When a thread T finishes executing a method on a monitor O, or when a thread T is signaled from a condition queue of O, T updates its version vector using the component-wise maximum of $T.v$ and $O.v$, i.e., $T.v = max(T.v, O.v)$.

Timestamps assigned using the above rules are called object-centric timestamps. Note that this scheme is preferred only if the number of synchronization objects is smaller than the number of threads. Considering that in a message-passing program, each thread usually has at least one port, we do not expect that this scheme will be frequently used for message passing programs. Fig. 10 shows object-centric timestamps assigned for the executions in Fig 9.

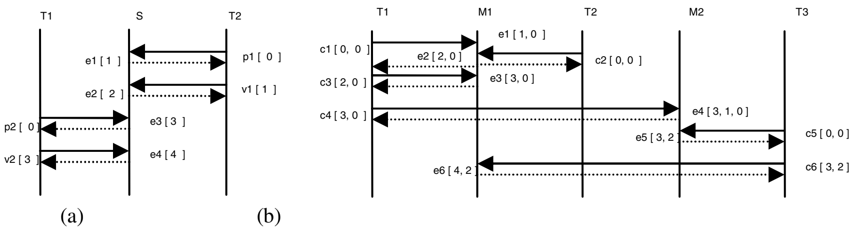


Fig. 10. Object-centric timestamps

Object-centric timestamps cannot be used to determine the happened-before relation between two arbitrary events. However, they can be used to determine the happened-before relation between two events if at least one of the events is a receiving event, which is sufficient for our purposes.

Proposition 2. Let X be an execution involving synchronization objects $O_1, O_2, \ldots, O_m$. Let Q be the *SYN*-sequence exercised by X. Assume that every event in Q is assigned an object-centric timestamp. Let r be a receiving event on O_i, and f a receiving event on O_j, where $1 \leq i, j \leq m$. Then, $e \rightarrow f$ if and only if $e.ts[i] \leq f.ts[i]$ and $e.ts[j] < f.ts[j]$.

Proposition 3. Let *X* be an execution involving synchronization objects $O_1, O_2, \ldots, O_m$. Let *Q* be the *SYN*-sequence exercised by *X*. Assume that every event in *Q* is assigned an object-centric timestamp. Let *r* be a receiving event on O_i, and *s* a sending event on O_j, where $1 \le i, j \le m$. Then, $r \rightarrow s$ if and only if $r.ts[i] \le s.ts[i]$.

In Fig. 10b, entry e_3 happens before entry e_4 since $e_3.ts[1] = e_4.ts[1]$ and $e_3.ts[2] < e_4.ts[2]$. Entry e_3 happens before call c_6 since $e_3.ts[1] = e_6.ts[1]$.

4.2 Race Detection

As we described in Section 3.6, we can characterize a program execution as a sequence of sending and receiving events. Intuitively, there exists a race between two sending events if they can synchronize with the same receiving event in different executions. In order to accurately determine all the races in an execution, the program's semantics must be analyzed. Fortunately, 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 solely based on the SYN-sequence of an execution, i.e., without analyzing the program's semantics.

Definition 1. Let *Q* be a *SYN*-sequence exercised by an execution of a concurrent program CP. Let *s* be a sending event and *r* 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, r*> in *Q* if *s'* and *r* can form a synchronization pair during some other execution of CP provided that all the events that happen before *s'* or *r* in *Q* are replayed in that execution.

Note that Definition 1 requires all events that can potentially affect *s'* or *r* in *Q* to be replayed and thus ensures the existence of *c'* or *e*, regardless of the program's implementation. As a result lead races can be identified solely based on information encoded in *Q*. In the remainder of this paper, a race is assumed to be a lead race unless otherwise specified.

Next, we define the notion of a race set of a receiving event. Let *Q* be a *SYN*-sequence. Let *r* be a receiving event and *s* a sending event in *Q* such that <*s, r*> is a synchronization pair. The race set of *r* in *Q*, denoted as *race(r, Q)* or *race(r)* if *Q* is implied, is the set of sending events in *Q* that have a race with <*s, r*>. Formally, $race(r, Q) = \{s' \in Q \mid \text{there exists a lead race between } s' \text{ and } <s, r>\}$.

The following proposition describes how to compute the race set of a receiving event.

Proposition 4. Let *Q* be a *SYN-sequence* exercised by a program execution. A sending event *s* is in the race set of a receiving event *r* 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) *s* and *r* are consistent with FIFO semantics (i.e., all the messages that were sent to the same destination as *s*, but were sent before *s*, have already been received before *r*).

Race Set Examples:

- *Asynchronous Message Passing*. The race set of each receive event in Fig. 2 is as follows: *race(r1) = {s2}, race(r2) = race(r3) = race(r4) = {}*. Note that *s3* is not in

the race set of *r1* because *s3* is sent to a different port and thus *s3* is not open at *r1*. For the same reason, *s4* is not in the race set of *r3*. Also note that *s4* is not in the race set of *r1*, because FIFO semantics ensures that *s1* is received before *s4*.

- *Synchronous message passing.* The race set of each receive event in Fig. 3 is as follows: *race(r1) = {s2}, race(r2) = { }, race(r3) = {s4}, and race(r4) = { }*. Since the alternative for *p2* is open whenever *T2* selects the receive-alternative for *p1*, the race set for *r1* contains *s2* and the race set for *r3* contains *s4*. On the other hand, since the alternative for *p1* was closed when *T2* selected the receive-alternative for *p2* at *r2*, the race set for *r2* does not contain *s3*.
- *Semaphores*. The race set of each completion event in Fig. 4 is as follows: *race(e1) = {p2}, race(e2) = race(e3) = race(e4) = { }*. Note that since *P()* was not in the open-list of *e2*, the race set for *e2* does not contain *p2*. 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.
- *Locks*. The race set of each completion event in Fig. 5 is as follows: *race(e1) = {l3}, race(e2) = race(e3) = race(e4) = race(e5) = race(e6) = {}*. Note that since *T2* owned lock *K* when the operations for events *e2*, *e3*, and *e4* were started, the race sets for *e2*, *e3*, and *e4* are empty. This represents the fact that no other thread can complete a *lock()* operation on *K* while it is owned by *T2*.
- *Monitors*. The race set of each entry event in Fig. 6 is as follows: *race(e1) = {c2}*, race(*e2*) = race(*e3*) = { }, race(*e4*) = {*c5*}, and race(*e5*) = race(*e6*) = { }. Sending event *c3* is not in the race set of *e2* since *c3* happened after *e2*. (Thread T2 entered monitor *m* at *e2* and executed a signal operation that caused *T1* to issue call *c3*.)

4.3 Computing Race Variants

Let *CP* be a concurrent program. Let *Q* be the *SYN*-sequence exercised by an execution of *CP*. Informally, a race variant of *Q* is the beginning part of one or more *SYN*-sequences of *CP* that could well have occurred but didn't, due to the way races were arbitrarily resolved during execution.

Definition 2. Let *Q* be a *SYN*-sequence and *V* be a race variant of *Q*. Let *partner(r, Q)* (or *partner(r, V)*) be the sending partner of a receiving event *r* in *Q* (or in *V*). Variant *V* is another *SYN*-sequence that satisfies the following conditions:

1. Let *r* be a receiving event in *Q*. If *r* is also in *V*, and if *call(r, Q) != call(r, V)*, then *call(r, V)* must be in *race(r, Q)*.
2. Let *f* be a sending or receiving event in *Q*. Then, *f* is not in *V* if and only if there exists a receiving event *r* in *Q* such that $r \rightarrow_Q f$ in *Q* and *partner(r, Q) != partner(r, V)*.
3. There exists at least one receiving event *r* in both *Q* and *V* such that *partner(r, Q) != partner(r, V)*.

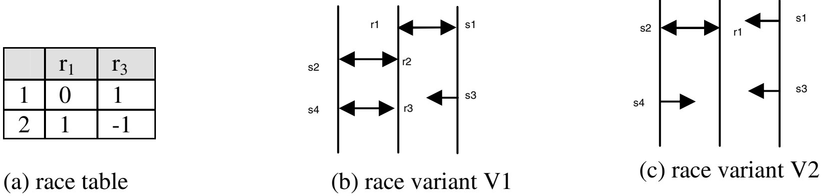
The first condition says that if we change the sending partner of a receiving event *r*, the new sending partner must be a sending event in the race set of *r*. The second condition says that if and only if we change the sending partner of a receiving event *r*, we remove all the events that happen after *r*. The third condition says that there must be at least one difference between *Q* and *V*.

Note that the second condition is a conservative approach to ensuring that a race variant is always feasible (i.e., it can be exercised by at least one program execution), regardless of the program's control and data flow. This condition is necessary since after the sending partner of a receiving event *r* is changed, all the events that happen after *r* could potentially be affected. That is, what happens after *r* might depend on the program's control and data flow. Note that this is a conservative approach since it removes events that happen after *r* even if they are not affected and are still feasible.

An algorithm for computing race variants of semaphore-based programs was described in [13]. This algorithm is easily adapted to the general execution model presented in this paper so we briefly describe it here. The race variants of a SYN-sequence are generated by building a so-called "race table", where each row of a race table corresponds to a race variant. The composition of a race table is described as follows. There is a column for each receiving event whose race set is non-empty. Let *r* be the receiving event corresponding to column *j*, *V* the race variant to be derived from row *i*, and *v* the value at row *i*, column *j*. Value *v* indicates how receiving event *r* is changed in variant *V*:

1. $v = -1$ indicates that *r* is removed from *V*
2. $v = 0$ indicates that the sending partner of *r* is left unchanged in *V*
3. $v > 0$ indicates that, in *V*, the sending partner of *r* is changed to the *v*-th event in *race(r)*, where the sending events in *race(r)* are arranged in an arbitrary order and the index of the first event in *race(r)* is *1*.

Note that when we change the sending partner of event *r*, we need to remove all the events that happened after *r* in the original execution. This is to be consistent with the second condition in Definition 2. The algorithm in [13] generates a race table whose rows contain all possible combinations of values for the receiving events.



	r_1	r_3
1	0	1
2	1	-1

(a) race table (b) race variant V1 (c) race variant V2

Fig. 11. Race variant examples

Fig. 11(a) shows the race table for the sample execution in Fig. 3. Recall that r_1 and r_3 are the only receiving events whose race sets are non-empty: $race(r_1) = \{s_2\}$ and $race(r_3) = \{s_4\}$. Fig. 11(b) shows the variant derived from the first row, where the sending partner of r_3 is changed from s_3 to s_4. Fig. 11(c) shows the variant derived from the second row, where the sending partner of r_1 is changed from s_1 to s_2, and event r_3 is removed since r_3 happened after r_1. See [13] for details about this algorithm.

5 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, and a test driver. The synchronization library provides classes for simulating semaphores, locks, monitors, and message passing with selective waits. The synchronization classes contain the necessary control for replaying variants and tracing SYN-sequences. The race variant generator inputs a SYN-sequence and generates race variants of the sequence as discussed in Sections 4. 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 presented in Section 2.

We wish to stress that RichTest does not require any modifications to the JVM or the operating system. Instead, the synchronization classes contain the additional control necessary 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 build 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 experiment in which RichTest was used to apply reachability testing to several components. The components chosen to carry out the experiment include: (1) BB – a solution to the bounded-buffer problem where the buffer is protected using either semaphores, an SC monitor, an SU monitor, or a selective wait; (2) RW – a solution to the readers/writers problem using either semaphores, an SU monitor, or a selective wait; (3) DP – a solution that uses an SU monitor to solve the dining philosophers problem without deadlock or starvation.

Table 1 summarizes the results of our experiment. The first column shows the names of the components. The second column shows the test configuration for each component. For BB, it indicates the number of producers (P), the number of consumers (C), and the number of slots (S) in the buffer. For RW, it indicates the number of readers (R) and the number of writers (W). For DP, it indicates the number of processes. The third column shows the number of sequences generated during reachability testing. To shed some light on the total time needed to execute these sequences, we observe that, for instance, the total execution time for the DP program with 5 philosophers is 7 minutes on a 1.6GHz PC with 512 MB of RAM.

Note that RichTest implements the reachability testing algorithm presented in [13]. The algorithm has a very low memory requirements, but it could generate duplicate SYN-sequences (i.e., exercise a given SYN-sequence more than once) for certain communication patterns. The reader is referred to [13] for more discussion on duplicates. In our experiment, the only case where duplicates were generated was for program BB-Semaphore. Since the number of duplicates may vary during different applications of reachability testing, we performed reachability testing on BB-Semaphore ten times and reported the average number of sequences exercised. We

note that the program has 324 unique sequences, and thus, on average, 64 sequences (or 18% of the total sequences) exercised during reachability testing were duplicates.

Table 1. Experimental Results

Program	Config	# Seqs.	Program	Config	# Seqs.	Program	Config	# Seqs.
BB-Select	3P + 3C + 2S	144	RW-Semaphore	2R + 2W	608	DP-Monitor SU	3	30
BB-Semaphore	3P + 3C + 2S	384 (avg.)	RW-Semaphore	2R + 3W	12816	DP-Monitor SU	4	624
BB-Monitor SU	3P + 3C + 2S	720	RW-Semaphore	3R + 2W	21744	DP-Monitor SU	5	19330
BB-Monitor SC	3P + 3C + 2S	12096	RW-Monitor SC	3R + 2W	70020			
			RW-Monitor SU	3R + 2W	13320			
			RW-Select	3R + 2W	768			

The results in Table 1 show that the choice of synchronization construct has a big effect on the number of sequences generated during reachability testing. SC monitors generate more sequences than SU monitors since SC monitors have races between signaled threads trying to reenter the monitor and calling threads trying to enter for the first time. SU monitors avoid these races by giving signaled threads priority over calling threads. Selective waits generated fewer sequences than the other constructs. This is because the guards in the selective waits are used to generate open-lists that reduce the sizes of the race sets.

6 Related Work

The simplest approach to testing concurrent programs is *non-deterministic testing*. The main problem with non-deterministic testing is that repeated executions of a concurrent program may exercise the same synchronization behavior. Most research in this area has focused on how to increase the chances of exercising different synchronization behaviors, and thus the chances of finding faults, when a program is repeatedly executed. This is typically accomplished by inserting random delays [23] or calls to a randomized scheduling function [17] into carefully selected program locations.

An alternative approach is *deterministic testing*, which is used to determine whether a specified sequence can or cannot be exercised. The main challenge for deterministic testing is dealing with the test sequence selection problem. A common method for selecting test sequences for deterministic testing is to derive a global state graph of a program (or of a model of the program) and then select paths from this

graph [21]. This method, however, suffers from the state explosion problem. Moreover, it is possible to select two or more paths that correspond to the same partial order, which is inefficient. Most research [10] [22] in this area has focused on how to address these two problems.

Reachability testing combines non-deterministic and deterministic testing. In [9], a reachability testing technique was described for multithreaded programs that use read and write operations. A reachability testing approach for asynchronous message-passing programs was reported in [19] and was later improved in [12]. These two approaches use different models to characterize program executions as well as different algorithms for computing race variants. Our work in this paper presents a general model for reachability testing. In addition, these approaches compute race variants by considering all possible interleavings of the events in a SYN-sequence. This is less efficient than our table-based algorithm where we deal with partial orders directly.

Recently, there is a growing interest in techniques that can systematically explore the state space of a program or a model of the program. The main challenge is dealing with the state explosion problem. The tools Java PathFinder I [8] and Bandera [5] first derive an abstract model of a Java program and then use model checkers such as SPIN to explore a state graph of the model. Techniques such as slicing and abstraction are used to reduce the size of the state graph. One problem these tools encounter is the semantic gap between programming languages and modeling languages, which makes some programming language features difficult to model. To overcome this problem, tools such as Java PathFinder II, VeriSoft [7] and ExitBlock [3] directly explore the state space of actual programs, i.e., without constructing any models. These tools use partial order reduction methods to reduce the chances of executing sequences that only differ in the order of concurrent events.

Reachability testing also directly explores the state space of actual programs. However, unlike VeriSoft, ExitBlock, and Java PathFinder II, our reachability testing algorithm deals with partial orders directly. In contrast, partial order reduction still generates total orders but tries to reduce the chances of generating total orders that correspond to the same partial order. In addition, the SYN-sequence framework used by reachability testing is highly portable. This is because the definition of a SYN-sequence is based on the language-level definition of a concurrency construct, rather than the implementation details of the construct. Our Java reachability testing tools do not require any modifications to the Java Virtual Machine (JVM) or the thread scheduler, and are completely portable. Our C/C++ tools for Windows, Unix, and Solaris, do not require any modifications to the thread scheduler either. In contrast, VeriSoft, ExitBlock and Java PathFinder II all rely on access to the underlying thread scheduler to control program execution, and the latter two tools also rely on a custom JVM to capture program states. As a result, these tools have limited portability.

7 Conclusion and Future Work

In this paper, we described a general model for reachability testing of concurrent programs. The main advantages of reachability testing can be summarized as follows:

- Reachability testing uses a dynamic framework to derive test sequences. This avoids the construction of static program models, which are often inaccurate and may be too large to build.
- If desired, reachability testing can systematically exercise all the behaviors of a program. This maximizes test coverage and has important applications in program-based verification.
- Reachability testing tools can be implemented in a portable manner, without modifying the underlying virtual machine, runtime-system or operating system.

We note that since reachability testing is implementation-based, it cannot by itself detect “missing sequences”, i.e., those that are valid according to the specification but are not allowed by the implementation. In this respect, reachability testing is complimentary to specification-based approaches that select valid sequences from a specification and determine whether they are allowed by the implementation [10].

We are continuing our work on reachability testing in the following directions. First, we are considering additional synchronization constructs, such as else/delay alternatives in selective wait statements. Second, exhaustive testing is not always practical due to resource constraints. Towards a more scalable solution, we are developing algorithms that can *selectively* exercise a set of SYN-sequences according to some coverage criteria. Third, there is a 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. Static analysis methods for model extraction have difficulty handling dynamic activities like creating threads and heap management. These things are easier to handle in a dynamic framework. 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 a concurrent program.

References

1. Ada Language Reference Manual, January 1983.
2. A. Bechini and K. C. Tai, Timestamps for Programs using Messages and Shared Variables, 18th Int’l Conf. on Distributed Computing Systems, 1998.
3. D. L. Bruening. Systematic testing of multithreaded Java programs. Master’s thesis, MIT, 1999.
4. R. Carver and K. C. Tai, "Replay and Testing for Concurrent Programs," IEEE Software, Vol. 8 No. 2, Mar. 1991, 66-74.
5. J. Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java Source Code, In Proceedings of the 22nd International Conference on Software Engineering, June, 2000.
6. C. J. Fidge, Logical Time in Distributed Computing Systems, IEEE Computer, Aug. 1991, pp. 28-33.
7. P. Godefroid. Model Checking for Programming Languages using VeriSoft. Proceedings of the 24th ACM Symposium on Principles of Programming Languages, pages 174-186, Paris, January 1997.

8. K. Havelund and Tom Pressburger. Model Checking Java Programs Using Java PathFinder. International Journal on Software Tools for Technology Transfer (STTT) 2(4): 366-381, April 2000.
9. G. H. Hwang, K. C. Tai, and T. L. Huang. Reachability testing: An approach to testing concurrent software. International Journal of Software Engineering and Knowledge Engineering, 5(4):493-510, 1995.
10. Koppol, P.V., Carver, R. H., and Tai, K. C., Incremental Integration Testing of Concurrent Programs, IEEE Trans. on Software. Engineering, Vol. 28, No. 6, June 2002, 607-623.
11. L. Lamport. Time, Clocks, and the Ordering of Events in a Dist. System, Comm. ACM, July 1978, pp. 558-565.
12. Yu Lei and Kuo-Chung Tai, Efficient reachability testing of asynchronous message-passing programs, Proc. 8th IEEE Int'l Conf. on Engineering for Complex Computer Systems, pp. 35-44, Dec. 2002.
13. Yu Lei and Richard H. Carver, "Reachability testing of semaphore-based programs", to be published in Proc. of the 28th Computer Software and Applications Conference (COMPSAC), September, 2004.
14. F. Mattern, Virtual Time and Global States of Distributed Systems, Parallel and Distributed Algorithms (M. Cosnard et al.), Elsevier Science, North Holland, 1989, pp. 215-226.
15. R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. Proc. of 3rd ACM/ONR Workshop on Parallel and Dist. Debugging, pp. 1-11, 1993.
16. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic race detector for multithreaded programs," Transactions on Computer Systems 15, 4 (November 1998), 391-411
17. S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In Proc. of the Second Workshop on Runtime Verification (RV), Vol. 70(4) of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.
18. K. C. Tai. Testing of concurrent software. Proc. of the 13th Annual International Computer Software and Applications Conference, pp. 62-64, 1989.
19. K. C. Tai. Reachability testing of asynchronous message-passing programs. Proc. of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems, pp. 50-61, 1997.
20. K. C. Tai, R. H. Carver, and E. Obaid, "Debugging concurrent Ada programs by deterministic execution," *IEEE Trans. Software Engineering*, 17(1):45-63, 1991.
21. R. N. Taylor, D. L. Levine, and Cheryl D. Kelly, Structural testing of concurrent programs, IEEE Transaction on Software Engineering, 18(3):206-214, 1992.
22. A. Ulrich and H. Konig, Specification-based Testing of Concurrent Systems, Proceedings of the IFIP Joint International Conference on Formal Description Techniques and Protocol Specification, Testing, and Verification (FORTE/PSTV '97), 1997.
23. C. Yang and L. L. Pollock. Identifying redundant test cases for testing parallel language constructs. ARL-ATIRP First Annual Technical Conference, 1997.