Reachability Testing : An Approach to Testing Concurrent Software *

Gwan-Hwan Hwang
Dept. of Computer Science
National Tsing-Hua University
Hsinchu, Taiwan, Rep. of China
e-mail: ghhwang@cs.nthu.edu.tw

Kuo-Chung Tai
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27659-8206, USA
e-mail: kct@csc.ncsu.edu

Ting-Lu Huang
Dept. of Computer Science & Information Engineer, National Chiao-Tung University
Hsinchu, Taiwan, Rep. of China
tlhuang@csie.nctu.edu.tw

Abstract

Concurrent programs are more difficult to test than sequential programs because of nondeterministic behavior. An execution of a concurrent program nondeterministically exercises a sequence of synchronization events, called a **synchronization sequence** *(or* **SYN-sequence***).* **Nondeterministic testing** *of a concurrent program P is to execute P with a given input many times in order to exercise distinct SYN-sequences and produce different results. In this paper, we present a new testing approach, called* **reachability testing**. *If P with input X contains a finite number of SYN-sequences, reachability testing of P with input X can execute all possible SYN-sequences of P with input X. We show how to perform reachability testing of concurrent programs using read and write operations. Also, we present results of empirical studies comparing reachability and nondeterministic testing. Our results indicate that reachability testing has advantages over nondeterministic testing.*

1 Introduction

Due to the unpredictable progress of concurrent processes and the use of nondeterministic statements, multiple (or repeated) executions of a concurrent program with the same input may produce different results. Therefore, a single execution of a concurrent program P with input X is insufficient to determine the correctness of P with input X. It is possible that some executions of P with input X produce correct results and other do not. This nondeterministic behavior makes concurrent programs more difficult to test than sequential programs.

*G. H. Hwang and T. L. Huang's work was supported in part by the ROC National Science Council under grant NSC81-0408-E009-08. K. C. Tai's work was supported in part by the USA National Science Foundation under grant CCR-8907807.

An execution of a concurrent program nondeterministically exercises a sequence of synchronization events, called a **synchronization sequence** (or **SYN-sequence**). The format of a SYN-sequence of a concurrent program depends upon the concurrent constructs used. A number of definitions of SYN-sequences for various concurrent constructs and languages were given in [9, 1, 10]. One testing approach, called **nondeterministic testing**, is to execute a concurrent program with a given input many times with the intention to exercise as many distinct SYN-sequences as possible. Another testing approach, called **deterministic testing**, is to force an execution of a concurrent program with a given input according to a given SYN-sequence. In this paper, we present a new testing approach, called **reachability testing**, which combines nondeterministic and deterministic testing and offers advantages over nondeterministic or deterministic testing.

This paper is organized as follows. Section 2 surveys several approaches to testing concurrent programs. Section 3 introduces the concept of reachability testing. Section 4 shows how to solve a major problem in reachability testing for concurrent programs using read and write operations. Section 5 describes details of reachability testing. Section 6 presents the results of empirical studies. Section 7 concludes this paper. This paper is based on the first author's master thesis [6].

2 Approaches to Testing Concurrent Software

This section describes nondeterministic and deterministic testing and their combinations. More details of these testing approaches can be found in [9]. Let P be a concurrent program. A SYN-sequence S is said to be feasible (valid) for P with input X if S is allowed

0-8186-6960-8/94 $04.00 © 1994 IEEE

by the implementation (specification) of P with input X. P is said to have a synchronization fault if a feasible (valid) SYN-sequence of P with input X is invalid (infeasible). A prefix of a feasible SYN-sequence of P with input X is said to be prefix- feasible for P with input X.

2.1 Nondeterministic Testing

Nondeterministic testing of a concurrent program P involves the following steps:

(1) Select a set of inputs of P,

(2) For each selected input X, execute P with X **many** times and examine the result of each execution.

Multiple, nondeterministic executions of P with input X may exercise different feasible SYN-sequences and thus may detect more errors than a single execution of P with input X. Two methods for increasing the chance of exercising different SYN-sequences of P are:

(a) control the scheduling of processes in the ready queue of the operating system, and

(b) insert delay statements into P with the amount of delay randomly chosen [2, 3].

Nondeterministic testing of P with input X has two major problems. First, some feasible SYN-sequence of P with input X may be executed many times. Second, some feasible SYN-sequences of P with input X may never be executed. If P with input X contains a finite number of SYN-sequences, nondeterministic testing of P with input X can never guarantee to accomplish exhaustive testing (i.e. execution of all possible SYN-sequences of P with input X).

2.2 Deterministic Testing

Deterministic testing of a concurrent program P involves the following steps:

(1) Select a set of tests, each of the form (X,S), where X and S are an input and a SYN-sequence of P respectively.

(2) For each selected test (X,S), force a **deterministic execution** of P with input X according to S. The forced execution determines whether S is feasible for P with input X.

Deterministic testing of P allows the use of SYN-sequences selected according to the implementation and specification of P. Also, deterministic testing of P can repeat previous executions of P; such testing is referred to as **replay**. However, deterministic testing has additional problems to solve. One major problem is the selection of pairs of inputs and SYN-sequences for a concurrent program. A number of methods for solving this problem were proposed [5, 14, 15, 12]. Details about deterministic testing of concurrent Ada programs and semaphore- or monitor-based concurrent programs were discussed in [1, 10].

2.3 Prefix-Based Testing and Replay

One possible combination of nondeterministic and deterministic testing, called **prefix-based testing** [9], is to execute a concurrent program P with input X and SYN-sequence S as follows:

(1) perform deterministic testing of P according to X and S until the end of S is reached,

(2) perform nondeterministic testing of P immediately after the end of S is reached. (If step (1) fails, then step (2) is skipped.)

If S is prefix-feasible for P with input X, such prefix-based testing is referred to as **prefix-based replay**. The purpose of prefix-based testing or replay is to start nondeterministic testing from a specific program state other than the initial program state.

3 Concept of Reachability Testing

This section explains the concept of our proposed testing approach, called **reachability testing**. Assume that S is the SYN-sequence of an execution of P with input X. Reachability testing of P with input X and SYN-sequence S involves the following steps:

(1) Use S to derive a set of prefixes of other feasible SYN-sequences of P with input X. Such prefixes, called **race-variants** of S, are derived by changing the outcomes of race conditions in S.

(2) Perform prefix-based replay of P with input X and each race-variant of S to derive and collect additional feasible SYN-sequences for P with input X.

(3) For each new SYN-sequence collected in (2), repeat (1) and (2).

If P with input X contains a finite number of SYN-sequences, reachability testing of P with input X can accomplish exhaustive testing of P with input X and thus can reach all possible states of P with input X and determine the correctness of P with input X.

In step (2) above, if the number of feasible SYN-sequences of P with input X is huge or infinite, we need to set a maximum limit on the length of each collected SYN-sequence. If we do so, the above procedure collects the set of feasible and prefix-feasible SYN-sequences of P with input X such that the length of each SYN-sequence is less than or equal to the maximum limit. For the sake of simplicity, in this paper we

discuss reachability testing in terms of feasible SYN-sequences only.

The above description of reachability testing is incomplete. Step (1), the derivation of race-variants for a SYN-sequence, depends upon the concurrent constructs allowed in the SYN-sequence. The next section shows how to perform step (1) for SYN-sequences of concurrent programs using read and write operations. Section 5 provides additional details about reachability testing other than step (1).

4 Derivation of Race-Variants for Concurrent Programs using Read and Write Operations

Section 4.1 defines the format of a SYN-sequence of a concurrent program using read and write operations. Such a SYN-sequence is called an **RW-sequence**. Also, section 4.1 gives examples of race-variants of an RW-sequence. Section 4.2 presents an algorithm for deriving race-variants of an RW-sequence.

4.1 RW-sequences and their Race-Variants

Let P be a concurrent program using read and write operations. Each shared variable in P is assigned a version number, which is initialized to zero and increased by one after each write operation on this variable [7]. An execution of P involves two types of synchronization events: read and write events. A **read event** is denoted as R(U,V), which refers to a read operation on variable U with version number being V. A **write event** is denoted as W(U,V), which refers to a write operation on variable U with the resulting version number being V. Assume that P contains processes P_1, P_2, $\cdots$, and P_n, where $n > 0$. A partially-ordered RW-sequence of P is denoted as (S[1], S[2], ..., S[n]). (In the following discussion, unless otherwise specified, an RW-sequence is assumed to be partially-ordered.) For $1 \leq i \leq n$, S[i] is a sequence of read and write events executed by process P_i and is denoted as ($S(i,1)$, $S(i,2)$, ..., $S(i,k_i)$), where $k_i \geq 0$ and S(i,j), $1 \leq j \leq k_i$, is a read or write event. The result of an execution of P with input X can be determined by X, P, and the RW-sequence of this execution. An RW-sequence is said to be feasible for P with input X if this RW-sequence is allowed by the implementation of P with input X.

Assume that the following RW-sequence S_1 is the SYN-sequence of an execution of P, which contains processes P_1 and P_2 and shared variables A and B.

$S_1[1]$ = (W(A,1), R(B,1), $\cdots$)
$S_1[2]$ = (W(B,1), R(A,1), $\cdots$)

Let "$\rightarrow$" and "$||$" denote the "happens before" and "concurrent" relations, respectively [8]. In S_1, W(A,1)$\rightarrow$R(A,1), W(B,1)$\rightarrow$R(B,1), W(A,1)$||$W(B,1), and R(B,1)$||$R(A,1). From S_1, we can determine that the first read and write operations on A have a race condition and so do the first read and write operations on B.

Since the first read and write operations in S_1 on A have a race condition. It is possible for P with input X to have an execution in which the read operation on A happens before the write operation on A (i.e., "R(A,1)" becomes "R(A,0)"). Therefore, there exist one or more feasible RW- sequences of P with input X having the following RW-sequence S_2 as a prefix

$S_2[1]$ = ()
$S_2[2]$ = (W(B,1), R(A,0))

For each of such feasible RW-sequences of P, process P_1 begins with a write operation on A, but the version number of this write operation depends upon the behavior of P after R(A,0). Thus, $S_2[1]$ is empty. S_2 is called a race-variant of S_1. Informally, a race-variant of S_1 is derived from S_1 by changing the outcome of a race condition in S_1 and deleting events in S_1 that happen after the modified event. A race-variant of S_1 is a prefix of another feasible RW-sequence of P with input X.

Since the first read and write operations in S_1 on B have a race condition, it is possible for P with input X to have an execution in which the read operation on B happens before the write operation on B (i.e., "R(B,1)" becomes "R(B,0)"). Thus, there exist one or more feasible RW-sequences of P with input X having the following RW-sequence S_3 as a prefix

$S_3[1]$ = (W(A,1), R(B,0))
$S_3[2]$ = ()

S_3 is also a race-variant of S_1. If S_2 or S_3 is used in prefix-based replay of P with input X, the resulting RW-sequence is different from S_1.

4.2 An Algorithm for Deriving Race-Variants of An RW-sequence

Let P be a concurrent program containing $n > 1$ concurrent processes P_1, P_2, $\cdots$, and P_n, and $m > 0$ shared variables V_1, V_2, $\cdots$, and V_m. Let S be a feasible RW-sequence of P with input X. To derive race-variants of S, we construct the **race-variant diagram** (or **RV diagram**) for S, which is a tree with each node representing a prefix or race-variant of S. The nodes in the RV diagram for S are generated by considering possible interleavings of read and write events in S. For

a node in the RV diagram for S, the path from the root node of the diagram to this node is a totally-ordered sequence of read and write events.

Each node N in the RV diagram for S contains the following two vectors:

index vector : $(I_1, I_2, \cdots, I_n)$, where I_j, $1 \leq j \leq n$, is the index of the last read or write operation in S[j] that is executed for generating node N.

version vector : $(E_1, E_2, \cdots, E_m)$, where E_k, $1 \leq k \leq m$, is the version number of variable V_k when node N is generated.

For the root node of an RV diagram, its index vector is (0,0,···,0) and its version vector (0,0,···,0). In the RV diagram for S, when a node N is generated from node N' due to a read or write operation, the index and version vectors of N are generated from those of N'. Also, the version number of the variable accessed by the read or write operation is compared with the version number of the corresponding event in S. If the two version numbers are the same, then N represents a totally-ordered prefix of S and is called a **prefix node**; otherwise, N represents a race-variant of S and is called a **race-variant node**. No nodes are generated from a race-variant node.

Two nodes in the RV diagram for S may have the same index and version vectors. If so, there are two possibilities:

- **both are prefix nodes**. This happens when, due to the existence of concurrent events, a prefix of S has two or more totally-ordered sequences. For a set of prefix nodes having the same index and version vectors, only one of them is used to generate child nodes. (To keep the RV diagram as a tree, prefix nodes having the same index and version vectors are not merged into one node.)
- **one is a prefix node and the other a race-variant node.** To illustrate this, consider the following RW-sequence S_4:

  $S_4[1]$ = (R(A,0),...)
  $S_4[2]$ = (W(A,1),...)

From the root node, we can generate a prefix node N_1 according to the sequence of a read on A and then a write on A, and a race-variant node N_2 according to the sequence of a write on A and then a read on A. For N_1, the read event is R(A,0), and for N_2, the read event is R(A,1). Although N_1 is a prefix node and N_2 a race-variant node, both N_1 and N_2 have same index vector, which is (1,1), and the same version vector, which is (1,0,...,0). (Assume that A is the first one in the list of shared variables.)

It is impossible that two or more race-variant nodes have the same index and version vectors. The reason is that since no child nodes are generated from race-variant nodes, each race-variant node in the RV diagram for S is generated by changing the outcome of just one race condition in S.

Algorithm RV_RW

```
input:  a feasible RW-sequence S of a concurrent
program using read and write operations.
output:  RV(S), a set of race-variants of S.
(1)Initially, the RV diagram for S contains the root
   node only.  For the root node, set its index vector
   to (0,0,···,0) and its version vector to (0,0,···,0)
   Also, label the root node "unmarked".
(2)Select an unmarked node, say N, in the RV diagram
   for S.  Assume that the index vector of N is
   (I1,I2,···,In).  For each j, 1 ≤ j ≤ n, if Ij < the
   length of S[j], construct a child node N' of N
   according to steps (a)-(f).  Then label N "marked".
   (a)Set the index vector of N' to that of N except
      that the jth element is Ij+1.
   (b)Set the version vector of N' to that of N.
   (c)Let Vk be the variable read or written in
      S(j,Ij+1) and E the version number of Vk in
      S(j,Ij+1).
   (d)If S(i,Ij+1) is a write operation on Vk,
      increase the kth element of the version of
      N' by 1.
   (e)Let E' be the kth element of the version
      vector of N'.
   (f)If E = E'
      then /* N and N' are prefix nodes */
        if the RV diagram for S already contains a node
           N'' with the same index and
           version vectors as N',
        then label N' "marked",
        else label N' "unmarked".
      else /* N' is a race-variant node */
       •label N' "marked" and "race-variant node".
       •construct the race-invariant S' associated
        with N' as follows:
        •for 1 ≤ k ≤ n, S'[k] = (S(k,1),···,S(k,Ik))
        •for S'[j], add element S'(j,Ij+1), which is
         R(Vk,E'), if S(j,Ij+1) is a read event on Vk,
         or W(Vk,E'), if S(j,Ij+1) is a write event
         on Vk.
(3)Repeat step (2) until all nodes in the RV diagram
   for S have marked.
```

To illustrate algorithm RV_RW, consider the following feasible RW-sequence Q of a concurrent program P with input X, where P contains processes P_1 and P_2 and shared variables A and B.

Q[1] = (R(A,0), R(B,0), W(A,2))
Q[2] = (W(A,1), W(B,1), R(A,2),W(B,2))

Fig. 1 shows the RV diagram for Q. Each node N in the diagram contains two vectors, the first one being N's index vector and the second one N's version vector. Each prefix node is denoted by a rectangle with solid lines and each race-variant node by a rectangle with dotted lines. The RV diagram for Q contains two pairs of prefix nodes with identical index and version

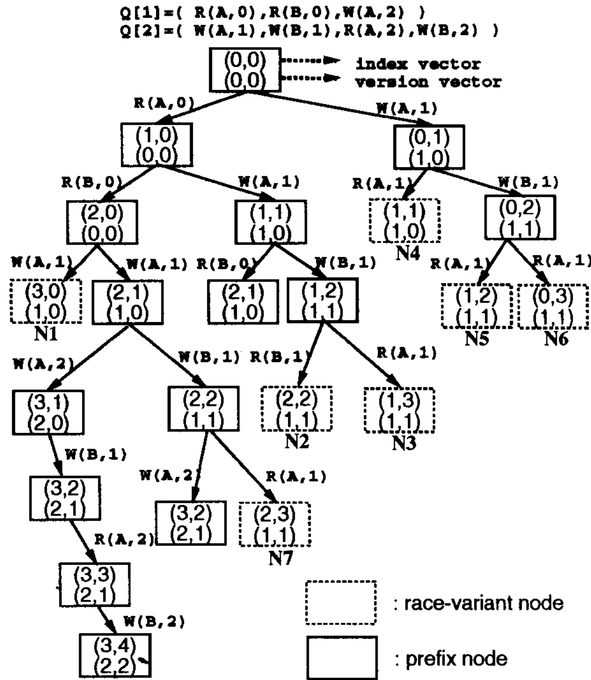

Figure 1: Parse tree of an array expression

vectors, one pair having (2,1) as the index vector and (1,0) as the version vector and the other pair having (3,2) as the index vector and (2,1) as the version vector.

The RV diagram for Q contains seven race-variant nodes, which are assigned names N_1 through N_7. The following RW-sequences Q_1 through Q_7 are the RW-sequences associated with nodes N_1 through N_7, respectively.

Q1[1] = (R(A,0), R(B,0), W(A,1))
Q1[2] = ()
Q2[1] = (R(A,0), R(B,1))
Q2[2] = (W(A,1), W(B,1))
Q3[1] = (R(A,0))
Q3[2] = (W(A,1), W(B,1), R(A,1))
Q4[1] = (R(A,1))
Q4[2] = (W(A,1))
Q5[1] = (R(A,1))
Q5[2] = (W(A,1), W(B,1))
Q6[1] = ()
Q6[2] = (W(A,1), W(B,1), R(A,1))
Q7[1] = (R(A,0), R(B,0))
Q7[2] = (W(A,1), W(B,1), R(A,1))

Theorem 1. Let S be a feasible RW-sequences of a concurrent program P with input X, and RV(S) be the set of race-variants of S produced by algorithm RV_RW. Assume that P contains processes P_1, P_2, $\cdots$, and P_n, where $n > 1$.

(a)For each element S' of RV(S),

(1)S' is not a prefix of S.

(2)S' differs from a prefix of S only in one event, which is the last event of S'[i] for some i, $1 \leq i \leq n$.

(3)S' is a prefix of at least one feasible RW-sequence of P with input X.

(b)For each feasible RW-sequence S'' of P with input X, where $S'' \neq$ S, S'' has at least one element of RV(S) as a prefix.

Proof. (a) It follows from the fact that S' is derived from S by changing the outcome of one race condition in S and deleting events in S that happen after the modified event. (b) Let S^* be the longest common prefix of S and S''. S and S'' have different outcomes of a race condition that occurs immediately after S^*. Thus, S^* is in RV(S). Q.E.D.

5 Reachability Testing of Concurrent Programs

The previous section shows an algorithm for generating race-variants of an RW-sequence. Similar algorithms can be derived for other types of SYN- sequences. The generation of race-variants is only one of the problems in reachability testing. Earlier, section 3 introduced the concept of reachability testing. This section provides additional details about reachability testing other than the generation of race-variants.

Assume that S is a feasible SYN-sequence of a concurrent program P with input X and S' is a race-variant of S. Also, assume that prefix-based replay of P with X and S' has generated another feasible SYN-sequence S'' of P with input X. To generate race-variants of S'', we consider only the race conditions in S'' that occur after S'. The reason is that the race conditions in S' have been considered in the generation of race-variants of S. To avoid the use of race conditions in S', a barrier symbol "@" is inserted to the end of S'. When S'' is used to derive race-variants, only the race conditions in S'' that occur after the "@" symbol are considered. After a race-variant S^* of S'' is generated, the "@" symbol is inserted to the end of S^*. Note that S^* has two "@" symbols. When S* is used to derive race-variants, only the race conditions that occur after the last "@" symbol are considered.

Below is an algorithm for reachability testing of a concurrent program P:

(1)Select a set of inputs of P.

(2)For each selected input X, perform the following steps.

(a)Let FS(P,X) be a set of feasible SYN-sequences of P with input X and RV(P,X) a set of race-variants of feasible SYN-sequences of P with input X. Initially, FS(P,X) and RV(P,X) are both empty.

(b)Execute P with input X to collect one or more feasible SYN-sequences. For each collected SYN-sequence, insert a "@" symbol at its beginning and add the resulting sequence to FS(P,X) as an unmarked sequence.

(c)For each unmarked sequence S in FS(P,X), derive its race-variants by modifying the outcomes of race conditions after the last "@" symbol in S. For each derived race-variant, if it is not already in RV(P,X), insert a "@" symbol at its end and add the resulting sequence to RV(P,X) as an unmarked sequence.

(d)For each unmarked sequence S' in RV(P,X), perform prefix-based replay of P with X and S' to collect one or more feasible SYN-sequences. For each collected SYN-sequence, if it is not already in FS(P,X), add it to FS(P,X) as an unmarked sequence.

(e)Repeat (c) and (d) until all sequences in FS(P,X) and RV(P,X) are marked.

The above algorithm for reachability testing is a high-level description. It needs refinement according to the concurrent constructs in P. It also needs improvement for performance consideration. As mentioned in section 3, when the sizes of FS(P,X) and RV(P,X) are huge or infinite, it is necessary to set limits on the sizes of FS(P,X) and RV(P,S).

Theorem 2: Assume that every execution of a concurrent program P with input X terminates. According to the above algorithm, reachability testing of P with input X derives and executes all feasible SYN-sequences of P with input X.

Proof. We show that after the completion of reachability testing of P with input X, FS(P,X) is the set of feasible SYN-sequences of P with input X. It is obvious that each sequence in FS(P,X) is a feasible SYN-sequences of P with input X. We need to prove that every feasible SYN-sequence of P with input X is in FS(P,X). If P with input X has only one feasible SYN-sequence, then this SYN-sequence is in FS(P,X). Consider the case that P with input X has two or more feasible SYN-sequences. Assume that S is a feasible SYN-sequence of P with input X and that S is not in FS(P,X). Let Q be a sequence in FS(P,X).

There exists a prefix S' of S such that S' is a race-variant of Q. Therefore, S' is in RV(P,X). Prefix-based replay of P with X and S' executes at least one feasible SYN-sequence, say Q', of P with input X. Q' is in FS(P,X). Since $Q' \neq$ S, there exists a prefix S'' of S such that S'' is a race-variant of Q' and has S' as a proper prefix (i.e., S'' is longer than S'). Hence, S'' is in RV(P,X). Prefix-based replay of P with X and S'' executes at least one feasible SYN-sequence, say Q'', of P with input X. Q'' is in FS(P,X).

By repeating the above argument, we can conclude that S is a race-variant of some sequence in FS(P,X) and that S is in RV(P,X). Prefix-based replay of P with X and S executes exactly S itself. Then S should be in FS(P,X). This contradicts our assumption that S is not in FS(P,X). Therefore, every feasible SYN-sequence of P with input X is in FS(P,X). Q.E.D.

6 Empirical Studies of Testing Concurrent Programs

This section describes two empirical studies using the following concurrent program, called Prod_Cons. This program implements the producer-consumer problem and contains processes A and B as producers and process C as a consumer. Each of processes A and B produces two items and deposit them to a queue. Process C withdraws four items from the queue and consumes them. A semaphore variable S is used to ensure mutual exclusion for accessing the queue. Prod_Cons is incorrect since it does not control the order in which items are deposited to and withdrawn from the queue.

```
process A
  do i= 1 to 2
     begin
       produce data
       WAIT S
         push data into the queue
       SIGNAL S
     end

process B
  do i= 1 to 2
     begin
       produce data
       WAIT S
         push data into the queue
       SIGNAL S
     end

process C
  do i= 1 to 4
     begin
       WAIT S
         pop data from the queue
       SIGNAL S
       consume data
     end
```

Since Prod_Cons uses P and V operations on a semaphore, a SYN-sequence of Prod_Cons is a **PV-sequence** [1], which contains events of the following four types:

(a) the start of execution of a P operation by a process on a semaphore,

(b) the end of execution of a P operation by a process on a semaphore,

(c) the start of execution of a V operation by a process on a semaphore,

(d) the end of execution of a V operation by a process on a semaphore.

For the sake of simplicity, we did not consider PV-sequences of program Prod_Cons in our empirical studies. Instead, we considered sequences of events of type (b) only. Such a SYN-sequence is referred to as a P-sequence [1]. The reasons for doing so are the following: (1) Program Prod_Cons correctly uses P and V operations on semaphore S to ensure mutual exclusion for the critical sections in processes A, B and C. (2) An event of type (b) in Prod_Cons implies entering a critical section. (3) P-sequences are simpler and shorter than PV-sequences.

A P-sequence of Prod_Cons is denoted as (C_1, C_2, $\cdots$), where C_i, $i > 0$, denotes the process executing the ith completed P operation on semaphore S. Thus, a P-sequence of Prod_Cons is exactly the sequence in which processes A, B and C enter their critical sections. A P-sequence of Prod_Cons contains two A's, two B's, and four C's. The number of distinct P-sequences of Prod_Cons is the number of different sequences of two A's, two B's, and four C's, and this number is $8!/(2!*2!*4!) = 420$. Some of these sequences, such as (C,C,C,C,A,A,B,B) and (A,C,C,C,C,A,B,B), are feasible, but **invalid** for Prod_Cons since they contain executions of withdraw operations when the queue is empty.

6.1 A Comparison between Nondeterministic and Reachability Testing

An empirical study was conducted to compare nondeterministic and reachability testing by running program Prod_Cons on a Sequent Symmetry machine with ten identical processors under normal system load. Since Prod_Cons contains three processes, its execution uses three processors, one for each process. As mentioned in section 2.1, nondeterministic testing can be done in different ways. The following two versions of nondeterministic testing were applied to Prod_Cons.

X : Number of executions
Y : repetition number

Y \ X	10	100	200	300	400	500	600	700	800	900	1000
1st	1	4	6	8	9	10	10	10	10	10	10
2nd	2	6	6	8	9	9	9	9	10	10	10
3rd	1	5	7	7	7	7	7	8	9	9	9
4th	1	4	5	7	7	8	8	9	10	10	10
5th	2	5	5	7	8	9	9	9	10	10	10
6th	1	3	6	6	6	7	9	9	10	10	10
7th	2	3	5	6	8	9	10	10	10	10	10
8th	2	5	5	7	8	8	8	9	10	10	10
9th	1	5	6	7	7	7	8	8	9	9	9
10th	1	3	6	7	9	9	10	10	10	10	10
avg	1.4	4.3	5.7	7.0	7.8	8.3	8.8	9.1	9.8	9.8	10.1

Table 1: Numbers of distinct RW-sequences of Prod_Cons collected by using NT_1

NT_1: nondeterministic testing without additional control.

NT_2: nondeterministic testing with inserted delay statements.

For each of NT_1 and NT_2, Prod_Cons was executed 1,000 times, and the number of distinct P-sequences collected was determined when the number of executions of Prod_Cons reached 10, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1,000, respectively. Also, the above process was repeated ten times and the average over the ten repetitions was computed. Tables 1 and 2 show these results. After 1,000 executions of Prod_Cons, the average number of distinct P-sequences collected by using NT_1 is 10.1, and that by using NT_2 is 14.9. Since the total number of feasible P-sequences of Prod_Cons is 420, only 2.4% and 3.5% of them were collected by NT_1 and NT_2, respectively. Furthermore, as shown in Tables 1 and 2, the number of distinct P-sequences collected by nondeterministic testing of Prod_Cons increases very slowly when the number of executions of Prod_Cons increases.

Since the Sequent machine has ten processors, the following two versions of reachability testing were applied to Prod_Cons:

RT_1: reachability testing with three processors: One of them was used to generate race-variants of P-sequences of Prod_Cons, and all three to execute Prod_Cons.

RT_2: reachability testing with ten processors: One processor was used to generate race-variants of P-sequences of Prod_Cons, and the other nine processors to execute three copies of Prod_Cons.

As expected, all of the 420 feasible P-sequences of Prod_Cons were generated by using each of RT_1 and RT_2.

X : Number of executions
Y : repetition number

Y \ X	10	100	200	300	400	500	600	700	800	900	1000
1st	8	15	15	15	15	15	15	15	15	15	15
2nd	6	11	13	13	14	14	14	14	14	14	14
3rd	7	13	13	14	14	14	14	15	15	15	15
4th	7	13	13	13	14	14	14	15	15	15	15
5th	8	11	12	12	14	15	15	15	15	15	15
6th	8	10	10	14	15	15	15	15	15	15	15
7th	9	12	12	13	13	13	15	15	15	15	15
8th	9	10	11	13	13	13	14	14	14	15	15
9th	5	10	14	14	14	14	14	14	14	15	15
10th	6	13	14	14	14	14	14	14	14	14	15
avg	7.3	11.8	12.7	13.6	14.0	14.1	14.3	14.5	14.5	14.7	14.9

Table 2: Numbers of distinct RW-sequences of Prod_Cons collected by using NT_2

One major concern with reachability testing is the amount of overhead for generating race-variants and performing prefix-based replay. The following table provides performance information:

	time (seconds)	results
NT_1	160	1,000 executions of Prod_Cons with about 10 distinct P-sequences collected
NT_2	240	1,000 executions of Prod_Cons with about 15 distinct P-sequences collected
RT_1	250	derivation and execution 420 distinct P-sequences of Prod_Cons
RT_2	150	derivation and execution 420 distinct P-sequences of Prod_Cons

From the above table, in 250 seconds or less, reachability testing derives and executes all 420 distinct feasible P-sequences of Prod_Cons, while nondeterministic testing executes less than 4% of distinct feasible P-sequences of Prod_Cons. Therefore, the overhead required by reachability testing is insignificant in light of the poor performance of nondeterministic testing. By comparing the execution time of RT_1 and RT_2, the speed-up due to the use of 10 processors (versus 3 processors) is 250/150 = 1.67.

6.2 Simulation of Nondeterministic Testing by Using Different Scheduling Policies

As mentioned in section 2.1, the effectiveness of nondeterministic testing may depend upon the scheduling of processes. We conducted an empirical study to investigate the impact of different scheduling policies on the effectiveness of nondeterministic testing. Since controlling the scheduling policy for an operating system was not allowed, we implemented a simulation of the following two scheduling policies:

- **round-robin**: after a process completes a time slice, it is kept at the end of the ready queue, and then the fist process in the ready queue becomes the next running process.
- **random selection with time slicing**: after a process completes a time slice, it is kept in the ready queue, and then one of the processes in the ready queue is randomly selected to be the next running process.

For each of the above two scheduling policies, the length of a time slice is not a constant. For a given maximum value T of a time slice, the actual value of a time slice is one of T/5, 2*T/5, 3*T/5, 4*T/5, and T, with equal probability.

The above two scheduling policies were applied to program Prod_Cons with the assumption that the initialization of a process takes 10,000 time units and the execution of each critical section 5,000 time unit. In our empirical study, T was set to 300, 500, 800, 1,000, 1,300, 1,500, 1,800, 2,000, 2,300, 2,500, 2,800, and 3,000 time units, respectively. For each given value of T, Prod_Cons was executed 50,000 times by using the round-robin policy, and the number of distinct P-sequences collected was determined when the number of executions of Prod_Cons reached 10, 100, 500, 1000, and so on. These results are shown in Table 3. Also, for each given value of T, Prod_Cons was executed 15,000 times by using the random selection policy. The results are shown in Table 4.

Our observations from Tables 3 and 4 are the following:

- The results of round-robin are consistent with those of NT_1 and NT_2 shown in section 6.1. This is expected since the round-robin policy was used by the Sequence machine for NT_1 and NT_2.
- Random selection with time slicing is more effective than round-robin in producing distinct P-sequences of Prod_Cons, and it eventually produces all 420 distinct P-sequences of Prod_Cons. The reason is that random selection of processes allows all possible interle[Bavings of events to occur, while round-robin tends to produce the same patterns of interleavings of events.
- From Table 4, 15,000 random-selection, nondeterministic executions of Prod_Cons are required to produce all 420 distinct P-sequences of Prod_Cons. According to the performance information for NT_1, such nondeterministic testing of Prod_Cons takes about 2,400 seconds. In contrast, reachability test-

X : Number of executions
Y : Time slice unit

Y \ X	10	100	500	1000	5000	10000	20000	30000	40000	50000
300	3	4	4	5	7	8	9	9	9	9
500	3	4	7	7	9	9	10	10	11	11
800	3	5	9	9	10	12	14	15	16	16
1000	3	6	8	9	12	12	14	15	16	17
1300	3	7	10	11	12	14	14	15	16	16
1500	4	8	9	10	11	14	15	15	15	15
1800	3	5	7	8	10	12	12	12	12	12
2000	2	6	6	8	8	8	9	10	10	10
2300	3	4	5	7	7	8	9	9	9	10
2500	4	7	9	9	9	9	10	10	10	10
2800	4	7	9	9	9	9	10	10	10	10
3000	4	5	9	10	10	10	10	10	10	10

Table 3: Numbers of distinct RW-sequences of Prod_Cons collected by using round-robin

X : Number of executions
Y : Time slice unit

Y \ X	10	100	200	400	600	800	1000	2000	3000	5000	10000	15000
300	9	77	126	191	237	273	305	383	404	417	420	420
500	10	72	121	197	243	276	299	358	394	409	418	420
800	10	77	125	200	247	282	301	366	394	413	419	420
1000	10	74	118	184	237	270	298	358	394	409	419	420
1300	10	78	129	189	239	271	299	366	399	416	420	420
1500	7	74	128	200	243	277	298	358	383	409	420	420
1800	9	77	136	208	249	284	306	370	389	410	420	420
2000	10	84	134	199	251	288	304	375	398	415	420	420
2300	10	75	134	200	249	282	311	382	404	415	420	420
2500	10	72	120	187	235	278	311	369	398	416	420	420
2800	10	72	120	187	235	278	311	369	398	416	420	420
3000	10	66	115	202	246	272	300	370	399	415	420	420

Table 4: Numbers of distinct RW-sequences of Prod_Cons collected by using random selection

ing of Prod_Cons takes only about 250 seconds to derive and execute all 420 distinct P-sequences of Prod_Cons. Therefore, reachability testing is about ten times faster than random- selection, nondeterministic testing for producing all distinct P-sequences of Prod_Cons.

7 Conclusions

In this paper we have presented a new approach, called reachability testing, to testing concurrent programs. If P with input X contains a finite number of SYN-sequences, reachability testing of P with input X can accomplish exhaustive testing of P with input X, and thus can determine the correctness of P with input X. We have shown how to perform reachability testing of concurrent programs using read and write operations. Based on the results of our empirical studies, reachability testing is significantly more cost-effective than nondeterministic testing for deriving distinct SYN-sequences of a concurrent program with a given input.

Reachability testing has similarities and differences with reachability analysis. Reachability analysis of a distributed program P is to derive the reachability graph of P, which contains all possible states and paths of P [13, 4] for all inputs of P. The reachability graph of P can be used to verify certain properties such as freedom from deadlock, livelock, or starvation [11]; it can also be used to select paths for testing or symbolic execution [14, 15]. Reachability testing of P with input X behaves like reachability analysis of P with input X since both derive all feasible SYN-sequences of P with input X. However, the former executes these SYN-sequences and produce outputs, while the latter derives these SYN-sequences and represents them in a graphical form. If the behavior of P has little or no dependence on inputs, reachability testing of P can verify the correctness of P.

Reachability testing is not just for accomplishing exhaustive testing, which is possible only for programs containing a finite number of feasible SYN-sequences. For a concurrent program having a huge or infinite number of feasible SYN-sequences, reachability testing can be applied to help the selection of a reasonable subset of SYN-sequences that is effective for fault detection. Also, the algorithm for reachability testing in section 5 needs to be improved to reduce time and space requirements.

Acknowledgements

The authors wish to thank Richard Carver for his helpful comments.

References

[1] R. H. Carver and K. C. Tai, *Replay and testing for concurrent programs.* IEEE Software," March 1991, 66-74.

[2] J. Gait, *A probe effect in concurrent programs.* Software-Practice and Experience, March 1986, 225-233.

[3] D. Helmbold and D. Luckham, *Debugging Ada tasking programs.* IEEE Software, Vol. 2, No. 2, March 1985, 47-57.

[4] G. J. Holzmann, *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[5] S. Y. Hsu and C. G. Chung, *A heuristic approach to path selection problem in concurrent program testing.* Proc. 3rd IEEE Workshop on Future Trends of Distributed Computing Systems, 1992, 86-92.

[6] G. H. Hwang, *A systematic parallel testing method for concurrent programs.* Master Thesis, Institute of Computer Science and Information Engineer, National Chiao-Tung Univ., Taiwan, 1993.

[7] T. J. LeBlanc and J. M. Mellor-Crummey, *Debugging parallel programs with instant replay.* IEEE Trans. Computers, Vol. C-36, No. 4, April 1987, 471-482.

[8] L. Lamport, *Time, clocks, and the ordering of events in a distributed systems.* Comm. ACM, Vol. 21, No. 7, July 1978, 45-63.

[9] K. C. Tai and R. H. Carver, *Deterministic execution testing and debugging of concurrent programs.* Proc. 1989 Pacific Northwest Software Quality Conference, 170-182. An extended version of this paper is published as Technical Report TR-93-14, Dept. of Computer Science, North Carolina State University, 1993.

[10] K. C. Tai, R. H. Carver, and E. E. Obaid, *Debugging concurrent Ada programs by deterministic execution.* IEEE Trans. Soft. Eng., Vol. 17, No. 1, Jan., 1991, 45-63.

[11] K. C. Tai, *Definitions and detection of deadlock, livelock, and starvation in concurrent programs.* Proc. 1994 Inter. Conf. Parallel Processing, 1994, Vol. II, 69-72.

[12] K. C. Tai and R. H. Carver, *Use of sequencing constraints in specification and testing of concurrent programs.* to appear in Proc. 1994 Int. Conf. Parallel and Distributed Systems, Dec. 1994

[13] R. N. Taylor *A General-Purpose Algorithm for Analyzing Concurrent Programs.* Communications of the ACM, Vol.26, No.5, 1983, 362-376.

[14] R. N. Taylor, D. L. Levine, and C. D. Kelly *Structural testing of concurrent programs.* IEEE Trans. on Software Eng., Vol. 18, No. 3, March 1992, 206-215.

[15] R. D. Yang and C. G. Chung *Path Analysis Testing of Concurrent Programs.* Information and Software Technology, Vol. 34. No. 1, Jan. 1992, 43-56.