# A New Algorithm for Reachability Testing of Concurrent Programs

Yu Lei
Dept. of Comp. Sci. and Eng.
The University of Texas at Arlington
Arlington, TX 76019
ylei@cse.uta.edu

Richard Carver
Department of Computer Science
George Mason University
Fairfax, VA 22030
rcarver@cs.gmu.edu

## Abstract

Concurrent programs exhibit non-deterministic behavior, which makes them difficult to test. One approach to testing concurrent programs, called *reachability testing*, generates test sequences automatically, and on-the-fly, without constructing any static models. This approach guarantees that every partially-ordered synchronization sequence of a program with a given input will be exercised exactly once. Unfortunately, in order to make this guarantee all existing reachability testing algorithms need to save and search through the history of test sequences that have already been exercised, which is impractical for many applications. In this paper, we present a new reachability testing algorithm that does not save the history of test sequences. This new algorithm guarantees that every partially-ordered synchronization sequence is exercised at least once for an arbitrary program and *exactly* once for a program that satisfies certain conditions. We also describe a reachability testing tool called RichTest. Our empirical studies with RichTest indicate that our new algorithm exercises every synchronization sequence exactly once for many applications.

## 1 Introduction

Because of its ability to increase computational efficiency, concurrent programming has become an important technique in modern software development. 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. (A SYN-sequence records the relative ordering of events that occur on a synchronization object such as a semaphore, monitor, or communication channel.) This non-deterministic behavior makes concurrent programs notoriously difficult to test.

*Reachability testing* is one approach to testing concurrent programs. A novel aspect of reachability testing is that it adopts a dynamic framework in which test sequences are generated automatically, and on-the-fly, without constructing any static program models. In this framework, the synchronization events that occur during a test run are recorded in an execution trace. At the end of the test run, the trace is analyzed to derive 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 exercise new behaviors, which are traced and then analyzed to derive more race variants, and so on. If every execution of a program P with input I terminates, and the total number of SYN-sequences is finite, then reachability testing will terminate and every partially-ordered SYN-sequence of P with input I will be exercised.

One potential problem with reachability testing is that some SYN-sequences may be exercised many times. To prevent this problem, all existing reachability testing algorithms save the history of SYN-sequences that have already been exercised and search the history before using a race variant. For large programs, the cost of saving a test history can be prohibitive both in terms of the space to store the history and the time to search it.

In this paper, we present a new reachability testing algorithm that does not save the test history. The main idea of our new algorithm is to enforce several constraints on how variants are generated so that a race variant is generated only if it can be used to exercise a SYN-sequence that has not been exercised before. This new algorithm is guaranteed to exercise every partially-ordered SYN-sequence at least once for an arbitrary program and exactly once for a program that satisfies certain conditions. We describe a Java reachability testing tool, called RichTest, which requires no modification to the Java JVM or to the operating system. Our empirical results indicate that the new algorithm exercises every SYN-sequence exactly once for many applications. To make our presentation concrete, we will show how to apply our new algorithm to asynchronous message-passing programs, i.e., programs in which processes synchronize/communicate by exchanging messages. However, we note that the authors reported a general reachability testing model in [2] that allows the new algorithm to be directly applied to other types of programs, including synchronous message-passing programs and shared-memory programs in which thread synchronize/communicate using semaphores, locks, and monitors.

The rest of this paper is organized as follows. The next section illustrates the reachability testing process. Section 3 presents an execution model for message-passing programs and introduces the notion of event equality. Section 4 provides an algorithm for generating race variants. Section

5 presents our new reachability testing algorithm. Section 6 describes the RichTest tool and reports some empirical results. Section 7 briefly surveys related work. Section 8 provides concluding remarks and describes our plan 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 six 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. A sequence of send and receive events is called an SR-sequence.

Fig. 1 also shows one possible scenario for applying reachability testing to the example program. Each SR-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. Note that in each SR-sequence, the portion above the dashed line is the race variant used to collect the sequence. 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 message race 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 T5 receives its first message from T6 instead of T4.

When variant V2 is used for prefix-based testing, sequence Q2 is collected. The variants and sequences derived after Q2 are shown in Fig. 1. Reachability testing stops after Q5 is recorded since Q0, Q1, Q2, Q3, Q4, and Q5 are all the possible SYN-sequences that can be exercised by this program.

For a formal description of the above process, the reader is referred to a reachability testing algorithm that we reported in [7]. The challenge for reachability testing is to ensure that every sequence is exercised once and only once. This is discussed in the remainder of this paper.

## 3 Preliminaries

### 3.1 SR-sequences

In this section, we describe how to model a program execution in which processes communicate and synchronize using asynchronous message passing. This model provides sufficient information for replaying an execution and for identifying the races in an execution. Also, this model is easily generalized to handle other synchronization constructs. In [2], the authors presented a general execution model for programs that use semaphores, locks, monitors, and asynchronous or synchronous message passing. Thus, the algorithms and techniques described in this paper can be applied without modification to programs that use any of the above constructs.

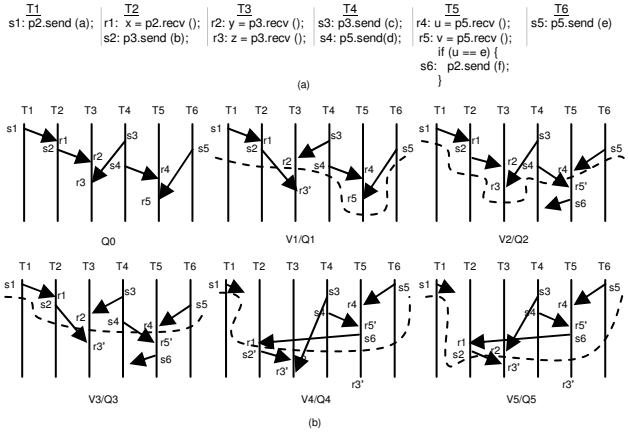



**Figure 1. The reachability testing process**

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 be received. A thread that executes a blocking receive operation blocks until a message is received. We assume that asynchronous ports (see section 2) use a FIFO (First-In-First-Out) message ordering scheme, which guarantees that the messages passed between any two threads are received in the order that they are sent.

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 form 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. Let Q be an SR-sequence and r a receive event in Q. We will use send(r, Q) to denote the send partner of r in Q.

Let *r* be a receive event and *s* be a send event in SR-sequence *Q* such that *<s, r>* is a synchronization pair. Suppose some other send event s' in *Q* could have been synchronized with r but wasn't, due to the arbitrary way in which race conditions were resolved during execution. Then s' is said to be in the *race set* of r. The authors have developed two schemes, a port-centric scheme and a thread-centric scheme, that can be used to identify races in an

execution [2]. For example, applying either scheme, we can identify the race set for each receive event in sequence Q0 in Fig. 1.b: race_set(r2) = {s3}, race_set(r4) = {s5}, race_set(r1) = race_set(r3) = race_set(r5) = {}. Due to space limitations, the reader is referred to [2] for details about these race identification schemes

### 3.2 Event Equality

In this section, we introduce our notion of event equality, which allows us to draw a correspondence between events that occur in different SR-sequences. We first define the prime structure of an event and an isomorphism between two prime structures, both of which are needed for defining the notion of event equality.

Assume that an event $e$ is generated by an execution instance of a statement $t$. Informally, the prime structure of $e$ contains all the events that could possibly affect the *beginning* of the execution of $t$. Note that if $e$ is a receive event, then the send partner $s$ of $e$ is not included in the prime structure of $e$. The reason is because $s$ only affects the end, rather than the beginning, of the execution of $t$.

**Definition 1:** Let $Q$ be an SR-sequence exercised by a program $P$ with input $I$. Let $e$ be an event in $Q$ that is exercised by a process $T$ in $P$. If $e$ is not the first event exercised by $T$, then let $f$ be the event that $T$ exercises immediately before $e$. Set *before(f)* contains the events that happen before $f$, and set $before^+(f) = before(f) \cup \{f\}$. Then, the prime structure of $e$ in $Q$, denoted as *p-struct(e, Q)* or *p-struct(e)* if $Q$ is implied, is empty if $f$ does not exist; otherwise, it is a prefix of $Q$ that contains all the events in $before^+(f)$.

As an example, consider sequence $Q_0$ in Fig. 1.b, where $p\text{-}struct(s_1)$ is empty, $p\text{-}struct(r_1)$ is empty, *p-struct(s2)* is the portion of Q0 consisting of the single event $r_1$, $p\text{-}struct(r_2)$ is empty, $p\text{-}struct(r_3)$ is the portion of Q0 consisting of the events *s1*, *r1*, *s2*, *r2*, $p\text{-}struct(s_3)$ is empty, $p\text{-}struct(s_4)$ is the portion of Q0 consisting of the single event $s_3$, *p-struct(r4)* is empty, *p-struct(s5)* is empty, and *p-struct(r5)* is the portion of Q0 consisting of events $s_3$, $s_4$ and $r_4$.

**Definition 2:** Let $Q$ and $Q'$ be two SR-sequences of a program $P$ with input $I$. Let $e$ be an event in Q and $e'$ an event in Q'. Then, *p-struct(e, Q)* and *p-struct(e', Q')* are isomorphic, denoted as *p-struct(e, Q) ~ p-struct(e', Q)'*, if there exists a one-to-one mapping $m$ from the events in *p-struct(e, Q)* to those in *p-struct(e', Q)'* such that if $<s, r>$ is a synchronization pair in *p-struct(e, Q)*, then $<m(s), m(r)>$ is a synchronization pair in *p-struct(e', Q')*..

Informally, two prime structures are isomorphic if their space-time diagrams are the same except their event labels.

**Definition 3:** Let $P$ be a message-passing program. Let $Q$ and $Q'$ be two SR-sequences of $P$ with input $I$. Let $e$ be an event in $Q$ and $e'$ an event in $Q'$. Events $e$ and $e'$ are equal, denoted as $e = e'$, if *p-struct(e, Q) ~ p-struct(e', Q')*.

In Fig. 1.b, we have used the same event label to name equals events in different SR-sequences. As an example, event r1 in Q0 equals event r1 in Q1, because both *p-struct(r1, Q0)* and *p-struct(r1, Q1)* are empty and thus are trivially isomorphic. In the remainder of this paper, we will consider equal events to be the same event and refer to them using the same label. We comment that the reason why the send partner $s$ of $e$ is not included in *p-struct(e)* is that otherwise, as soon as $e$ is synchronized with a different send event, $e$ becomes a *different* event. As a result, we would not be able to express the notion of a race set, where the *same* receive event can be synchronized with *different* send events.

## 4 Computing Race Variants

In Section 4.1, we define the notion of a race variant. In Section 4.2, we briefly explain an algorithm for computing the race variants of an SR-sequence.

### 4.1 Race Variant

Let P be a message-passing program. Let Q be the SR-sequence exercised by an execution of P. Intuitively, a race variant of Q can be derived by changing the send partner of one or more receive events in a way that satisfies the following constraints: Whenever the send partner of a receive event is changed: (1) the new send partner must be in the race set of r in Q; and (2) all the send and receive events that happen after r must be removed from Q.

Note that the second constraint is needed to ensure that a race variant is feasible (i.e., the variant can be exercised by at least one program execution), regardless of the program's control and data flow. This is because a change to the send partner of a receive event r in Q could potentially affect all the events that happened after r. In particular, if P takes a different path after a different message is received at $r$, then some events that happened after r in Q may no longer happen along the new path.

As an example, consider race variant V1 in Fig. 1.b, which is derived from SR-sequence Q0 by changing the send partner of r2 to s3, as s3 is in race_set(r2), and then removing r3, which is the only event that happens after r2.

A formal definition of a race variant is presented below.

**Definition** 4: Let Q be an SR-sequence. A race variant V of Q is another SR-sequence that satisfies the following conditions:

- There exists at least one receive event r in Q and V such that send(r, Q) ≠ send(r, V).
- Let r be a receive event in Q and V such that send(r, Q) ≠ send(r, V). Then, send(r, V) must be in race_set(r, Q).
- Let e be a send or receive event in Q. Then, e is in V if and only if for any receive event r in Q such that $r \rightarrow_Q e$, then send(r, Q) = send(r, V).

The first condition says that the send partner of at least one receive event needs to be changed. The second and third conditions are consistent with constraints (1) and (2) above, respectively.

One interesting phenomenon, called event recollection, is that some of the events that have been removed from V (to satisfy the third condition) may be generated again during prefix-based testing with V. Below we present two important properties related to event recollection and equality.

**Proposition 1:** Let *Q* be an SR-sequence and *V* a race variant of *Q*. Let *e* be an event in *Q* that was removed from *V*. Then, event *e* can be recollected during prefix-based testing with *V* if and only if there is no receive event in *p-struct(e, Q)* whose send partner was changed in *V*.

The above proposition is true because (1) if there is no receive event in *p-struct(e, Q)* whose send partner was changed, then *p-struct(e, Q)* is repeated in V; and (2) if *V* is forced to be exercised at the beginning of prefix-based testing with V, then *p_struct(e, Q)* will also be repeated during prefix-based testing with *V*.

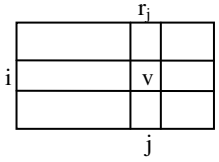
**Proposition 2**: Let *Q* be an SR-sequence. If a send event *s* in *Q* was removed from a race variant *V* of *Q*, then *s* cannot be recollected during prefix-based testing with *V*.

The above proposition can be derived from Proposition 1 for the following reasons: (1) Since event *s* in Q was removed from a race variant *V* of *Q*, there must exist a receive event *r* that happens before *s* in *Q* and whose send partner has been changed; (2) Any receive event that happens before a send event must also exist in the prime structure of the send event.

### 4.2 An Algorithm for Computing Race Variants

In [7], we reported an algorithm for computing the race variants of a semaphore-based execution. In [2], we reported a general model for reachability testing, which encapsulates the differences between the various types of synchronization constructs. The general model allows the algorithm in [7] to be applied to a message-passing execution. Below, we present a high-level description of this algorithm.

The algorithm for computing race variants builds a "race table" for a given SR-sequence Q. Each row of the race table for Q can be used to derive a unique, partially-ordered race variant of Q. The race table of Q consists of a column for each receive event in Q whose race set is non-empty. Let *r* be the receive event corresponding to column *j*. Let V be the race variant to be derived from row *i* and *v* be the value in row *i*, column *j*.




Value *v* indicates how receive event *r* in Q is changed to create variant V:

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

As an example, the race table for Q0 in Fig. 1.b is shown below. Variants V1, V2, and V3 are derived from rows 1, 2, and 3 in the table, respectively.

| r4 | r2 |
|---|---|
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

One approach to building a race table is to enumerate all combinations of the possible values of *v* for the receive events and then remove the invalid combinations. Invalid combinations are combinations that are not consistent with Definition 4. The algorithm reported in [7] builds the race table more efficiently by preventing a large number of invalid combinations from being generated in the first place. Due to space limitations, we refer the reader to [7] for details about the algorithm.

We stress that every row in a race table represents a unique, partially ordered variant [7]. Therefore, our algorithm deals with partial orders directly – test sequences are never totally ordered.

## 5 A New Reachability Testing Algorithm

In order to reduce test effort while maximizing test coverage, it is desirable to exercise every partially-ordered SR-sequence exactly once during reachability testing. However, if a newly derived race variant V is a prefix of an SR-sequence Q that has already been exercised, then prefix-based testing with V could exercise Q again. To deal with this duplication problem, all existing reachability testing algorithms need to save the history of SR-sequences that have already been exercised. A newly derived variant is used for prefix-based testing only if it is not a prefix of any SR-sequence that is in the test history. For large programs, the cost of saving a test history can be prohibitive both in terms of the space to store the history and the time to search it. As a result, the scalability of the existing reachability testing algorithms is limited.

In this section, we present a new reachability testing algorithm that does not save the test history. The main idea is to enforce several constraints that will prevent a race variant from being generated in the first place if prefix-based testing with this variant could cause duplicate SR-sequences to be exercised. The new algorithm guarantees that every SR-sequence is exercised at least once for an

arbitrary program and *exactly* once for a program that satisfies certain conditions.

### 5.1 A Graph-Theoretic Perspective

To help understand our new algorithm, we consider the reachability testing problem from a graph-theoretic perspective. Let P be a message-passing program. All possible SR-sequences that could be exercised by P with input I can be organized into a directed graph G, which we refer to as a Sequence/Variant graph or simply an S/V graph. Each node *n* in G represents an SR-sequence that could be exercised by P with input I. An edge *e* from node *n* to node *n'* represents a race variant of *n* and indicates that *n'* could be exercised by prefix-based testing with *e*. Note that a node *n* may have multiple outgoing edges which are labeled by the same variant of *n*. The reason is that prefix-based testing with a race variant forces the variant to be exercised at the beginning of a test run and then lets the run continue *non-deterministically*, and the non-deterministic portion can exercise different sequences in different test runs. For example, in Fig. 2, node Q2 has two outgoing edges that represent the two sequences that can be exercised during prefix-based testing with variant V4.

**Definition 5:** There is a *race difference* for a receive event *r* w.r.t. two SR-sequences if *r* exists in both sequences but has different send partners in these sequences.

Note that Def. 1 focuses on differences that are directly caused by message races. If we assume that the order in which threads synchronize and communicate is the only source of non-determinism, then any difference between two SR-sequences can be traced back to a race difference. For this reason, we will only be interested in race differences and will refer to a race difference as a difference unless stated otherwise.

**Theorem 1**: Let P be a message-passing program. Let G be the S/V graph of P with input I. Then, G is strongly connected.

**Proof**: We show that given two arbitrary nodes *n* and *n'* in G, where $n \neq n'$, there exists a path from *n* to *n'* and another path from *n'* to *n*. By symmetry, we only need to show that there exists a path from *n* to *n'*. In the following, we demonstrate how to construct such a path step by step.

Let $D = \{r \mid r$ is a receive event that exists in both *n* and *n'* but *r* has different send partners in *n* and *n'*.$\}$. Note that *D* is not empty, as otherwise *n* and *n'* would be the same sequence. There must exist an outgoing edge *e* of node *n* in which the send partner of every receive event $r \in D$ is changed to match the send partner of *r* in *n'*. Let *n''* be the destination node of *e*. If *n''* = *n'*, then there is a path from *n* to *n'* which only consists of edge *e*. Otherwise, the longest common prefix between *n''* and *n'* is longer than that between *n* and *n'*, This is because every receive event $r \in D$ now has the same send partner *s* in *n''* and *n'*, and every event that happens before *r* must be the same in *n''* and *n'* (otherwise *r* and *s* could not have the same prime structures in n" and n' and thus could not both exist in *n''* and *n'*). This process can be repeated until we reach *n'*.

From a graph-theoretic perspective, the goal of reachability testing is to construct a spanning tree of an S/V graph. Note that since an S/V graph is strongly connected, reachability testing can start from an arbitrary node. Also note that during reachability testing, each variant is used to conduct only one test run. Therefore, in a spanning tree constructed during reachability testing no two edges are labeled with the same variant.

In Fig. 2, the left side shows the S/V graph of program P in Fig. 1.a, and the right side shows a spanning tree that can be generated during reachability testing of P.

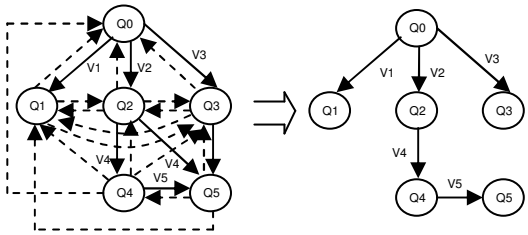



**Figure 2: The S/V-graph of the program in Fig. 1 and a spanning tree of the S/V-graph**

Now it becomes clear that the purpose of saving the test history during reachability testing is to avoid visiting the same node more than once. Therefore, the main challenge for our new algorithm is to avoid visiting the same node (i.e., exercising the same sequence) more than once *without* saving a list of the nodes that have already been visited.

### 5.2 Path Constraints

Let G be the S/V graph of a program P with input I. The main idea of our new algorithm is as follows: If we can find several constraints on the paths through G such that given two arbitrary nodes *n* and *n'* in *G*, there is exactly one acyclic path from *n* to *n'* that satisfies these constraints, then we can construct a spanning tree of G by only generating those paths satisfying these constraints. In this section, we will define three such path constraints. Note that since it is possible that no path from one node to another satisfies these constraints, enforcing these constraints alone will not construct a spanning tree. This will be discussed in Section 5.3. In Section 5.4, we will show how these constraints are implemented in our new algorithm.

Let *n* and *n'* be two arbitrary nodes in G. Let $H = n_1 e_1 n_2 e_2 \ldots n_m$, where $n = n_1$ and $n' = n_m$, be a path from node *n* to node *n'* in G. We consider each edge $e_i$ to represent a transformation of node $n_i$ into node $n_{i+1}$. (Note that we will use node/sequence and edge/variant interchangeably.) This transformation is realized by changing the send partners of some of the receive events in node $n_i$ to derive a race variant $e_i$ and then performing prefix-based testing with variant $e_i$. In this manner, sequence *n* is eventually transformed into *n'*. The path

constraints we present below impose restrictions on how the send partner of a receive event can be changed at each edge $e_i$ of path $H$.

**C1: The send partner of a receive event can be changed at most once along path *H*.**

Constraint C1 ensures that $H$ is an acyclic path. Assume that there exists a cycle in H. Let $n_i e_i n_{i+1} \ldots n_j$ be such a cycle, where $1 \le i, j \le m$, $i \ne j$, and $n_i = n_j$. Note that $e_i$ changes the send partner of at least one receive event $r$ in $n_i$. Assume that the send partner of $r$ is $s$ in $n_i$. Since $n_i = n_j$, the send partner of $r$ must have been changed back to $s$ by some edge $e_k$, $i < k < j$. This is impossible, since the send partner of $r$ is already changed in $e_i$, and it can be changed at most once along path H.

Constraint C1 also implies that the send partner of a receive event cannot be changed to a send event other than its send partner in $n'$. In other words, if the send partner of a receive event $r$ is changed in an edge $e_i$, then the new send partner of $r$ must match the send partner of $r$ in $n'$. The reason is because otherwise, in order to reach $n'$, the send partner of $r$ would have to be changed again sometime later, which is impossible. In the following, we will say "a difference is reconciled" between $n_i$ and $n'$, $1 \le i < m$, if the send partner of a receive event $r$ is changed in edge $e_i$ to match the send partner of $r$ in $n'$.

**C2: Differences must be reconciled in happens-before order.**

Let $r_1$ and $r_2$ be two receive events in node $n_i$, $1 \le i < m$, such that $r_1$ happens before $r_2$. Assume that both events also exist in node $n'$ but have different send partners in $n'$. Note that in this case, $r_1$ must not be in *p-struct*$(r_2, n_i)$. Otherwise, since *p-struct*$(r_2, n_i)$ is not repeated in $n'$, $r_2$ cannot exist in $n'$. Also note that these two differences cannot be reconciled at the same time (i.e., in the same edge), since when we reconcile the difference with $r_1$ in one edge, $r_2$ will be removed in that edge (see constraint (2) on deriving race variants in section 4.1). Therefore, constraint C2 dictates that if the difference with $r_1$ is reconciled in edge $e_i$ and the difference with $r_2$ is reconciled in edge $e_j$, then $i < j$.

Let $f_i$ be the longest common prefix between $n_i$ and $n'$, $1 \le i < m$, where an event is included in $f_i$ if and only if it exists in both $n_i$ and $n'$ and it has the same send partner in $n_i$ and $n'$. We show that constraints C1 and C2 together ensure that if $i < j$, then $f_i$ is a proper prefix of $f_j$. First we show that C1 ensures that $f_i$ is a prefix of $f_j$. Assume that there is an event $a$ in $f_i$ that does not exist in $f_j$. Then, there must exist a receive event $r$ such that $r$ happens before $a$ in $n_i$, and the send partner of $r$ is changed in some edge $e_k$, $i < k \le j$. This is impossible, because $r$ is in $f_i$, which means that the send partner of $r$ in $n_i$ is the same as that in $n'$, and the send partner of $r$ is already changed in $e_k$.

Next we show that C2 ensures that $f_i$ is a proper prefix of $f_j$. Observe that each edge $e$ reconciles at least one difference. It suffices to show that the reconciliation of each difference adds one more receive event to the longest common prefix. Let $r$ be a receive event in a node $n_k$, $i < k \le j$. Assume that the send partner of $r$ in $n_k$ is different from that in $n'$. Also assume that the send partner of $r$ in $n_k$ was changed in edge $e_k$. By C2, all the receive events that happen before $r$ must have the same send partners in $n_k$ as in $n'$; otherwise, the send partner of $r$ cannot be changed in $e_k$. This means that $r$ must be in $f_{k+1}$.

**C3: Each edge must reconcile as many differences as possible.**

Constraint C3 means that if a difference can be reconciled by an edge without violating C1 and C2, then the difference should be reconciled by that edge.

The following lemma shows that given any two nodes $n$ and $n'$ in an S/V graph, there is at most one path from $n$ to $n'$ that satisfies all three constraints.

**Lemma 1:** Let $G$ be an S/V graph. Let $n$ and $n'$ be two arbitrary nodes in $G$. Then, there exists at most one path from $n$ to $n'$ that satisfies all the three constraints C1, C2 and C3.

**Proof:** Earlier we have shown that C1 can only be satisfied by an acyclic path from $n$ to $n'$. Next we show that constraints C2 and C3 can be satisfied by at most one acyclic path. We proceed by contradiction. Assume that there exist two acyclic paths $H1$ and $H2$ from $n$ to $n'$ that satisfy C2 and C3. Let $n''$ be the first node where the two paths branch off. Let $e_1$ and $e_2$ be two outgoing edges of $n''$ in paths H1 and H2, respectively. Then, there is at least one difference reconciled in one of the two edges but not in the other. Without loss of generality, let $r$ be a receive event such that its send partner is changed in $e_1$ but not in $e_2$. Then, by C2, all the differences that happened before $r$ must have already been reconciled before node $n''$. Therefore, $r$ could have been reconciled in $e_2$, which means that $e_2$ does not satisfy C3, leading to a contradiction.

To illustrate the above constraints, consider the example program in Fig. 1. In the S/V graph in Fig. 2, there exists an edge from $Q_1$ to $Q_0$ that completes a cycle. Note that send(r2, $Q_0$) = s2 and send(r2, $Q_1$) = s3. Therefore, the edge from $Q_1$ to $Q_0$ must change the send partner of *r2* from *s3* to *s2*. However, we note that the send partner of r2 has already been changed once in V1. Therefore, the edge from Q1 to Q0 will be prevented by C1.

As another example, consider path $Q_0Q_3Q_5$ in Fig. 2. This path is excluded from the spanning tree because r1 happens before r2 in $Q_0$, however the difference with r1 is reconciled in the second edge and the difference with r2 is reconciled in the first edge, which violates C2.

Finally, consider path $Q_0Q_2Q_3$ in Fig. 2. This path is excluded from the spanning tree because the difference with r2 could have been reconciled in V2, but was not, which violates C3.

### 5.3 Tangled Cycles

Unfortunately, in certain cases, there may be no path from one node to another that satisfies all three constraints. Fig. 3.a shows a program involving threads T1, T2, T3, and T4. As usual, we assume that each thread *Ti* has port $p_i$. Fig. 3.b shows all the possible SR-sequences of the program. Fig. 3.c shows the S/V graph of the program. Assume that node $Q_0$ is chosen as the start node. The dashed edges in Fig. 3.c will be suppressed by constraint C1. Then, the only path left in the S/V graph from $Q_0$ to $Q_2$ is $Q_0Q_1Q_2$, which does not satisfy constraint C2, since r1 happens before r2 in $Q_0$ but the difference with r2 is reconciled in V1 before the difference with r1 is reconciled in V2.

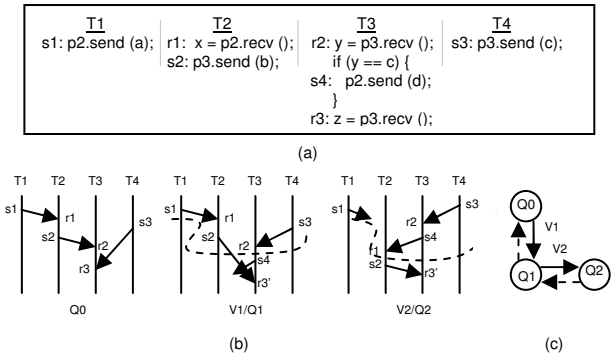



**Figure 3. A tangled cycle**

We observe that since r1 happens before r2 in $Q_0$, constraint C2 states that the difference with r1 should be reconciled before the difference with r2. However, in order to reconcile the difference with r1, we must first collect event s4, which is the send partner of r1 in Q2. But this can only happen after we have reconciled the difference with r2, since s4 occurs only after r2 receives the message sent by s3. This gives rise to a cycle of reconciliation, called a tangled cycle, which makes it impossible to find a path satisfying constraint C2.

Our solution to this problem is to allow constraint C2 to be violated if and only if such a tangled cycle occurs. In the following discussion, we characterize such a cycle. As usual, let n and n' be two arbitrary nodes in an S/V graph. Let H be a path from n and n'. We will refer to a receive event r as a to-be-reconciled event w.r.t. n and n', or simply a to-be-reconciled receive event when n and n' are implied, if r exists in both n and n' but has different send partners in them.

**Definition 6**: Let *r* and *r'* be two *to-be-reconciled* receive events w.r.t. *n* and *n'*. Then, (1) *r* should be *weakly reconciled before r'* w.r.t. *n* and *n'*, denoted as $r \rightarrow_{wrb(n, n')} r'$ or simply $r \rightarrow_{wrb} r'$ when *n* and *n'* are implied, if *r* happens before *r'* in *n*; (2) *r* should be *strongly reconciled before r'* w.r.t. *n* and *n'*, denoted as $r \rightarrow_{srb(n, n')} r'$ or simply $r \rightarrow_{srb} r'$ when *n* and *n'* are implied, if *r* happens before the send partner of *r'* in *n'*.

Note that the weakly-reconciled-before relation is due to constraint C2. By "weakly", we mean that this relation can be broken in certain cases. In contrast, the strongly-reconciled-before relation is an absolute requirement of the process of reconciliation, which says that certain events must occur before a difference can be reconciled. By "strongly", we mean that this relation must be maintained all the time.

**Definition 7**: A tangled cycle w.r.t. *n* and *n'* consists of a sequence of receive events $r_1, r_2, \ldots, r_n, r_{n+1} = r_1$ in *n* and *n'* such that $r_1 \rightarrow_{wrb(n, n')} r_2 \rightarrow_{srb(n, n')} r3 \rightarrow_{wrb(n, n')} \cdots \rightarrow_{srb(n, n')} r_{n+1}$, where $1 \le i \le n$.

Note that in a tangled cycle, *weakly-reconciled-before* and *strongly-reconciled-before* links must alternate. For example, in Fig. 3, there exists a tangled cycle w.r.t. $Q_0$ and $Q_2$: $r_1 \rightarrow_{wrb} r_2 \rightarrow_{srb} r_1$.

If there exists a tangled cycle between *n* and *n'*, then it is impossible to reconcile the receive events involved in the cycle in the happens-before order. In this case, the cycle must be broken at a *weakly-reconciled-before* link $r_i \rightarrow_{wrb} r_j$, i.e., we should allow $r_j$ to be reconciled before $r_i$, which means that constraint C2 may not be maintained. Note that the cycle cannot be broken by reconciling the difference with $r_i$. For this reason, we call $r_i$ as a non-cycle-breaking event and $r_j$ a cycle-breaking event. Also note that if a cycle has more than one *weakly-reconciled-before* link, then it could be broken in different ways, each of which will result in a different path from *n* to *n'*. In order to ensure that only one of these paths is generated, we add a restriction that a tangled cycle can only be broken at one of those links. To make our choice consistent, we place a total order on all the weakly-reconciled-before links based on the event ID of their cycle-breaking events, i.e., a link $r_1 \rightarrow_{wrb} r_1'$ is smaller than another link $r_2 \rightarrow_{wrb} r_2'$ if the event ID of $r_1'$ is smaller than the event ID of $r_2'$. (We assume that every event has a unique integer ID.) Then, a tangled cycle can only be broken at the smallest *weakly-reconciled-before link*.

**E1**: Constraint C2 is allowed to be violated in order to break a tangled cycle.

In Fig. 3, the only way to break the cycle is to reconcile $r_1$ before $r_2$. This allows path $Q_0Q_1Q_2$ to be generated during reachability testing. Note that in this scenario, constraint C2 is not maintained due to exception E1.

As discussed in Section 5.4, our new reachability testing algorithm needs to solve the following problem. Let *Q* and *T* be two SR-sequences. Let *r* and *r'* be two *to-be-reconciled* events w.r.t. *Q* and *T*. Assume that *r* happens before *r'* in *Q*, i.e., $r \rightarrow_{wrb} r'$. Now we want to determine whether $r \rightarrow_{wrb} r'$ is a weakly-reconciled-before link in a tangled cycle. Let *R* be the set of all the *to-be-reconciled* receive events w.r.t. *Q* and *T*. We can organize all the events in *R* into a graph *G*, where each node represents a *to-be-reconciled* receive event. There are two types of edges in *G*. Let $r_1$ and $r_2$ be two receive events in *G*. There is a

*weak* edge from $r_1$ to $r_2$ if $r1 \rightarrow_{wrb} r2$, and a *strong* edge from *r1* to *r2* if $r1 \rightarrow_{srb} r2$. Therefore, $r \rightarrow_{wrb} r'$ is involved a tangled cycle if and only if *r* can be reached from *r'* in *G*, which can be accomplished by a modified depth-first search algorithm that restricts that weak and strong edges must alternate.

### 5.4 The Algorithm

Fig. 4 shows algorithm *GenerateVariants*. Given an SR-sequence Q and the race variant that was used to collect Q, this algorithm generates a subset of the variants of Q. In algorithm *GenerateVariants*, the receive events in sequences Q and V are colored either white, black or gray. The color of a receive event *r* in *V* is inherited by the equivalent event r in Q. (Recall that V is a prefix of Q so each event in V is also in Q.) Receive events that are in Q but not in V are colored white. The color of receive event *r* restricts how *r* can be changed when deriving variants: *white* indicates that the send partner of *r* can be changed; *black* indicates that the send partner of *r* cannot be changed; and *gray* indicates that the send partner *r* may be changed, but only under certain conditions. More details about assigning and using colors are provided below.

Next we use the example programs in sections 2 and 5.3 to illustrate how algorithm *GenerateVariants* enforces the three path constraints C1, C2, and C3, and the exception rule E1. A more detailed explanation on this can be found in [14].

*C1: The send partner of a receive event can be changed at most once.*

The method for enforcing this constraint is simple. In step 3, if the send partner of a receive *r* in *Q* is changed to derive a race variant *V'*, then the color of *r* in *V'* is set to black (due to statement (*) in *GenerateVariants*). In step 2, black receive events are excluded from the heading of the race table, which prevents the send partners of black receives from being changed again. For example, in Fig. 1.b, variant *V1* (the portion of *Q1* that is above the dashed line) was derived by changing the send partner of *r2* from *s2* to *s3*. Therefore, the color of *r2* in *V1* will be black, which is inherited by *r2* in *Q1*. Thus, *r2* will be excluded from the heading of the race table, which prevents the send partner of *r2* from being changed, when we derive the race variants of *Q1*.

Note that it is possible for a black receive *r* in sequence *Q* to be removed when deriving variant *V'* of Q. (If the send partner of a receive event in *Q* is changed to generate variant *V'*, then the events in *Q* that happen after the changed event are not included in *V'* (see section 4.1).) If *r* were recollected in an SR-sequence *Q'* derived from *V'*, the color of *r* in *V'* would not be inherited, which means that *r*'s color would be white in *Q'*. As a result, the send partner of *r* could be changed again. This scenario violates constraint C1 and is prevented by statement (**), which sets the color of a receive event *r'* to black in *V'* if (i) *r'* happened before *r* in *V'* and (ii) *r'* is not in *p-struct(r, V')*. This means that the send partner of *r'* cannot be changed in sequences derived from *V'*. By Proposition 1, the only way to recollect *r* is to change the send partner of such a receive event as *r'*. Therefore, the fact that we cannot change the send partner of r' means that *r* cannot be recollected, which prevents the above scenario.

*C2: Differences must be reconciled in happens-before order, with the following exception E1: Constraint C2 is allowed to be violated in order to break a tangled cycle.*

Constraint C2 is enforced by coloring some receive events as gray. Essentially, if the send partner of a receive event *r* is changed to derive variant *V'*, then the color of *r'* is set to gray if *r'* happens before *r* in *V'* but *r'* is not in *p-struct(r, V')*. The send partner of a gray receive cannot be changed unless they complete a tangled cycle that was broken earlier due to Exception E1. This is because otherwise the difference with *r'* was reconciled after the difference with *r* was reconciled, which violates C2. Note that the color of a receive event *r'* is not changed if *r'* is in *p-struct(r, V')*. This is because if the send partner of *r'* is ever changed, then *r* will not exist (since changing *r'* changes *r*'s prime structure). Thus, it is impossible to reconcile the difference with *r* before the difference with *r'*. This implies that C2 can never be violated and thus the color of r' does not need to be changed.

Consider the program in Fig. 3. Variant *V1* is derived by changing the send partner of *r2*. Note that *r1* happens before *s2* in *V1* but is not in *p-struct(r2, V1)*. Thus, the color of *r1* will be set to gray in *V1* by statement (***) in *GenerateVariants*. Note race variant *V2* is derived from *Q1* by changing the send partner of *r1*, which is a gray receive event. The reason why *V2* is not discarded is because changing *r1* completes the tangled cycle consisting of *r1* and *r2*, which was broken earlier when the send partner of *r2* was changed in *V1*.

*Constraint 3: Each edge in the S/V graph must reconcile as many differences as possible.*

This constraint is enforced in an implicit manner. Let *n* and *n'* be two nodes in the S/V graph. If a difference w.r.t. *n* and *n'* could be reconciled by an edge *e*, but is not, then this difference will be marked and can never be reconciled afterwards. Therefore, no path from *n* to *n'* can be generated if it contains any such edge as *e*, since there exists at least one difference that cannot be reconciled between *n* and *n'*. This is equivalent to saying that if we generate a path from *n* to *n'*, then each edge in the path must reconcile as many differences as possible.

The above idea is implemented by step 1 in *GenerateVariants*, which removes "old" send events from the race sets of "old" receive events. A send or receive event in an SR-sequence *Q* is "old" if it also appears in the variant *V* used to collect *Q*. For example, consider SR-sequence $Q_2$ in Fig. 1.b. Note that events *r2* and *s3* are old

events because they appear in both *V2* (the prefix of Q2 which is above the dashed line) and *Q2*. Therefore, *s3* will be removed from the race set of *r2*, which means that if we did not change the send partner of *r2* from *s2* to *s3* in V2, then we would never be able to do that in the future.

The complexity of algorithm *GenerateVariants* is dominated by the second step. The original table-based algorithm is in $O(N_v * N_e^3)$, where $N_v$ is the number of race variants of $Q$, and $N_e$ the number of events in $Q$. The modification for black receives does not change the complexity. In the modification for gray receives, it takes $O(N_e)$ to determine if the change completes a tangled cycle. Therefore, the complexity of this modification is $O(N_v * N_e^2)$. Note that this modification can be performed after the first modification was completed. Therefore, the complexity of the second step, and thus the complexity of the algorithm, is $O(N_v * N_e^3)$. As mentioned below, reachability testing exercises every SR-sequence exactly once for many applications. Since this algorithm is used to generate variants for each sequence, in the average case, the complexity of the entire testing process is $O(N_Q * N_v * N_e^3)$, where $N_Q$ is the number of all possible SR-sequences.

```
VariantList GenerateVariants (SR-sequence Q, Variant V)
// Q was collected during prefix-based testing with V
begin
// 1: prune "old" send events from the race sets of "old" receive
//    events in Q.
   for each receive event r in V (and thus in Q too) do
     race_set(r, Q) = race_set(r, Q) - race_set(r,V)

// 2: generate the race variants of Q, variants(Q), using the
// table based algorithm in section 4.2 with the following
// modifications for handling black and gray receive events:
a. modification for black receives: exclude black receives from
    the heading of the race table to prevent their send partners
    from being changed;
b. modification for gray receives: the send partner of a gray
    receive r cannot be changed unless the change reconciles
    the last deifference of a tangled cycle C that was broken at
    an early point and r is the non-cycle-breaking event of the
    smallest weakly-reconciled-before link in C

// 3: set the colors of receive events in the variants
  for each variant V' in variants(Q) do
     for each receive r whose send partner is changed do
       r.color = black; … (*)
       for each receive r' that is not in p-struct(r, V') do
         if r' happens before r in V'
         then r'.color = black; … (**)
         else if r' happens before send(r, Q) in V'
                 and r'.color ≠ black
              then r'.color = gray; … (***)
  return variants(Q);
end
```

**Figure 4. Algorithm *GenerateVariants***

We note that the new reachability testing algorithm may exercise the same SR-sequence more than once if some tangled cycles are interconnected. Two tangled cycles are interconnected if one or more receive events are involved in both cycles. This is because unlike for a regular cycle, it is difficult to determine whether a change completes a tangled cycle that was broken at an early point. However, our empirical studies indicate that interconnected cycles are not common.

**Theorem 2:** Let $P$ be a message-passing program and $I$ an input of $P$. Let $\sum$ be the set of all the feasible SR-sequences of $P$ with input $I$. Then, our new reachability testing algorithm based on algorithm *GenerateVariants* has the following properties:

I. It exercises every SR-sequence in $\sum$ at least once.

II. It exercises every SR-sequence in $\sum$ exactly once if there is no interconnected cycles w.r.t. any two SR-sequences in $\sum$.

The formal proof of Theorem 2 is presented in Appendix.

## 6 Empirical Results

We implemented our reachability testing algorithms in a prototype tool called RichTest. RichTest provides a Java class library that contains a race variant generator class, a test driver class, a class for tracing and replaying SR-sequences, and synchronization classes for simulating semaphores, locks, monitors, and message passing with selective waits. RichTest is implemented entirely in Java and does not require any modifications to the JVM or the operating system. We are applying this same approach to build portable reachability testing tools for multithreaded C++ programs that use thread libraries written for Windows, Solaris, and Unix.

As a proof-of-concept, we conducted an experiment in which RichTest was used to apply reachability testing to several programs: (1) BB – a solution to the bounded-buffer problem where the buffer is protected using either semaphores, a Signal-and-Continue (SC) monitor, a Signal-and-Urgent-Wait (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 name of the program. The second column shows the test configuration for each program. 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 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. There were no interconnected cycles found for any of these programs. As a result, RichTest was able to exercise every SYN-

sequence of these programs exactly once. 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.

| Program | Configuration | # of Seqs |
|---|---|---|
| BB-Select | 3P + 3C + 2S | 144 |
| BB-Semaphore | 3P + 3C + 2S | 324 |
| BB-MonitorSU | 3P + 3C + 2S | 720 |
| BB-MonitorSC | 3P + 3C + 2S | 12096 |
| RW-Semaphore | 2R + 2W | 608 |
| RW-Semaphore | 2R + 3W | 12816 |
| RW-Semaphore | 3R + 2W | 21744 |
| RW-MonitorSC | 3R + 2W | 70020 |
| RW-MonitorSU | 3R + 2W | 13320 |
| RW-Select | 3R + 2W | 768 |
| DP-MonitorSU | 3 | 30 |
| DP-MonitorSU | 4 | 624 |
| DP-MonitorSU | 5 | 19330 |

**Table 1. Experimental Results**

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.

# 7 Related Work

One approach to testing concurrent programs is non-deterministic testing, which executes the same program with the same input many times and hope that faults will be exposed by one of these repeated executions [3][10]. The main problem with this approach is that because of lack of control, some SYN-sequences may be exercised many times while others are never exercised.

An alternative approach is deterministic testing, which forces test runs to exercise selected SYN-sequences. The SYN-sequences are usually selected from a global state graph of a program (or of a model of the program) [11][12]. This approach suffers from the state explosion problem. Moreover, it may select totally-ordered SYN-sequences that are different linearizations of the same partial order, which is inefficient.

Reachability testing combines non-deterministic and deterministic testing. In [5] a reachability testing algorithm for multithreaded programs that use shared variables was described. In [9] a reachability testing algorithm for asynchronous message-passing programs was reported, which was later improved in [6]. The authors have recently reported two reachability testing algorithms, one for semaphore-based programs [7] and the other for monitor-based programs [8], and a general model for reachability testing [2]. All the existing algorithms need to save the history of test sequences and thus have limited scalability.

Recently, there is a growing interest in techniques that can directly explore the state space of actual programs without constructing any models. Tools such as Java PathFinder [13], VeriSoft [4] and ExitBlock [1] use partial order reduction methods to reduce the chances of executing totally-ordered synchronization sequences that only differ in the order of concurrent events. In contrast, our reachability testing algorithm deals with partial orders directly – no totally-ordered sequences are ever generated. 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 reachability testing tool requires no modification to the thread scheduler and is completely portable, while Java PathFinder, VeriSoft, and ExitBlock all require access to the thread scheduler to control program execution, and have limited scalability.

# 8 Conclusion and Future Work

In this paper, we have described a new algorithm for reachability testing of concurrent programs. For many practical applications, this new algorithm can exercise every possible SYN-sequence exactly once, without saving the history of test sequences. This represents a significant reduction in memory requirements and thus allows reachability testing to be applied to large programs. We note that since reachability testing is implementation-based, it cannot by itself detect "missing SYN-sequences", i.e., sequences 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.

We are continuing our work on reachability testing in the following three directions. First, exhaustive testing is not always practical due to resource constraints. To seek a trade-off between test effort and test coverage, we are developing algorithms that can selectively exercise a set of SYN-sequences according to some test coverage criteria. Second, we are addressing the test oracle problem. Reachability testing frequently executes a large number of sequences, which makes it impractical to manually inspect the output of all the test executions. At present, properties such as freedom from deadlock and assertion violations can be checked automatically in RichTest. We plan to build new RichTest components so that advanced temporal properties can be checked automatically as well. 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. 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] D. L. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[2] R. Carver and Y. Lei, A general model for reachability testing of concurrent programs, to appear in Proc. of Intl. Conf. on Formal Engineering Methods, 2004.

[3] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithread Java program test generation. *IBM Systems Journal*, Vol. 41(1), pp. 111-125, 2002.

[4] 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

[5] 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 Eng. 5(4):493-510, 1995.

[6] Y. Lei and K. C. Tai, Efficient reachability testing of asynchronous message-passing programs, Proc. 8th IEEE Int'l Conf. on Engineering for Complex Computer Systems, pp. 35-44, 2002.

[7] Y. Lei and R. Carver, Reachability testing of semaphore-based programs, Proc. of COMPSAC, 2004.

[8] Y. Lei and R. Carver, Reachability testing of monitor-based programs, Proc. of Software Engineering and Applications, 2004.

[9] 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.

[10] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proceedings of the Second Workshop on Runtime Verification (RV)*, Vol. 70(4) of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[11] K. C. Tai. Testing of concurrent software. *Proc. of the 13th Annual International Computer Software and Applications Conference*, pp. 62-64, 1989

[12] 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.

[13] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder – Second Generation of a Java Model Checker. *In Proc. of Post-CAV Workshop on Advances in Verification*, 2000.


# Appendix

## 1 Correctness Proof of Theorem 2

**Notations**: Let $P$ be a message-passing program. Let $Q$ be an SR-sequence of $P$ with input $X$. Let *Send(Q)* be the set of send events in $Q$. Let *send(r, Q)* be the send partner of a receive event $r$ in $Q$. Let *Recv(Q)* be the set of receive events in $Q$. Let $Sync(R, Q) = \{<s_1, r_1>, ..., <s_n, r_n>\}$, where $R = \{r_1, ..., r_n\}$ is a subet of *Recv(Q)*, and $s_i = send(r_i, Q)$. Let *Min(R, Q)*, where $R$ is a subset of *Recv(Q)*, be a subset of $R$ consisting of receive events that are minimal w.r.t. the happens-before relation of $Q$, i.e., $Min(R, Q) = \{r \in R \mid$ *no receive event* $r' \in R$ *such that* $r' \rightarrow_Q r\}$. Let *Q1* and *Q2* be two SR-sequences, and let $Diff(Q1, Q2) = \{r \in Recv(Q1) \cap Recv(Q2) \mid send(r, Q1) \neq send(r, Q2)\}$, and $Cut(Q1, Q2) = Min(Min(Diff(Q1, Q2), Q1), Q2)$.

### 1.1 Part I of Theorem 2

Fig. 5 presents a procedure called *Guided-RT*, which selects a race variant at each step to guide the testing process until a given SR-sequence is eventually exercised. We show that the race variant selected by *Guided-RT* is generated by algorithm *GenerateVariants* at each step and that procedure *Guided-RT* terminates.

```
Procedure Guided-RT
// Input: A message-passing program P, an input X of P, and
//        an SR-sequence T of P with input X
//Output: T is exercised by an execution of P with X
begin
  execute P with X non-deterministically to collect an
    SR-sequence Q0
  Q = Q0;
  while Cut(T, Q) is not empty do
  begin
    // 1: select a subset of events to reconcile
    // a. Ready-To-Change: receives that can be reconciled in
    // the happens-before order
    RTC = { r ∈ Cut(T, Q) | ∀r' ∈ Q: r' →Q r ⇒ send(r', Q) =
            send(r', T) };
    // b. Ready-To-be-Untangled: receives that need to be
    // untangled
    RTU = {};
    for each receive event r ∈(Cut(T, Q) - RTC) do
      if (∀ r' ∈ Diff(T, Q), r' →Q r ⇒ both r' and r are involved
          in a tangled cycle w.r.t. Q and T)
          AND
          (r is the smallest cycle-breaking event of a tangled
           cycle w.r.t Q and T)
      then RTU = RTU + r;
    // c. Ready-To-be-Reconciled: the union of RTC and RTU
    RTR = RTC ∪ RTU;

    // 2. select a race variant which reconciles the differences
    // with the receives in RTR
    find a variant V in variants(Q) so that
     (1) Sync(RTR, V) = Sync(RTR, T);  ... (*)
     (2) Sync(Recv(V) - RTR, V) = Sync(Recv(V) - RTR, Q)

    // 3: perform prefix-based testing
    perform a prefix-based testing of P with X using V, and let
Q
        be the new sequence
    end
```

**Lemma 1**: In *Guided-RT*, the variant $V$ at each iteration is generated by algorithm *GenerateVariants*.

Proof: Let $Q^i$, $V^i$, $RTR^i$, $RTC^i$, $RTU^i$ be the SR-sequence, variant, *RTR*, *RTC*, and *RTU* at iteration $i$. Let $r \in RTR^i$. We show that (1) *send(r, T)* exists in $Q^i$ and $send(r, T) \in race(r, Q^i)$; (2) if $r$ exists in $V^{i-1}$, then (2.1) *send(r, T)* does not exist in $V^{i-1}$ (so that *send(r, T)* is not pruned from the race set of $r$) (2.2) *r.color != black*; (2.3) if *r.color = gray*, then changing the send partner of $r$ to *send(r, T)* reconciles the last difference in a tangled cycle in which $r$ is the non-cycle-breaking event of the smallest weakly-reconciled-before link.

**Case 1**: First we show *send(r, T)* exists in $Q^i$. This is because otherwise, there must exist a receive event $r'$ in $T$ so that $r' \rightarrow_T send(r, T)$ (and thus $r$) but receives different messages in $T$ and $Q^i$. This implies that $r$ cannot be in $Cut(T, Q^i)$, leading to a contradiction.

Next we show that (a) $r$ does not happen before *send(r, T)* in $Q^i$ and (b) letting $r'$ be the receive event with which *send(r, T)* is synchronized in $Q^i$, then $r$ happens before $r'$.

**Case 1.a**: Assume that $r$ happens before *send(r, T)* in $Q^i$. Since $send(r, Q^i) \neq send(r, T)$, *send(r, T)* cannot exist in $T$ (by Proposition 2), leading to a contradiction.

**Case 1.b**: Assume that $r'$ happens before $r$. (Note that $r'$ and $r$ must belong to the same process and thus cannot be concurrent.) Since in $T$, *send(r, T)* is synchronized with $r$, i.e., no longer with $r'$, $send(r', T) \neq send(r, T) = send(r', Q^i)$. Thus $r$ can't be in $Cut(T, Q^i)$, leading to a contradiction.

**Case 2.1**: Assume that *send(r, T)* also exists in $V^{i-1}$. We show that $r \in RTR^{i-1}$, which means that $r$ will be reconciled at iteration $i - 1$. Therefore, $r$ cannot be in $Cut(T, Q^i)$, leading to a contradiction.

First, we show that $r \in Cut(T, Q^{i-1})$. Assume otherwise, and consider two cases: (2.1.a) $r$ is not in $Min(Diff(T, Q^{i-1}), T)$; (2.1.b) $r$ is in $Min(Diff(T, Q^{i-1}), T)$.

**Case 2.1.a**: There must exist a receive event $r'$ such that $r' \rightarrow_T r$ and $r' \in Min(Diff(T, Q^{i-1}), T)$. If $r'$ is in *p-struct(r, T)*, then $r$ could not exist in $Q^{i-1}$ (by event equality), leading to a contradiction. If $r'$ is not in *p-struct(r, T)*, then $r' \rightarrow_T send(r, T)$. Since $send(r', T) \neq send(r', Q^{i-1})$, *send(r, T)* could not exist in $Q^{i-1}$ (by Proposition 2), also leading to a contradiction.

**Case 2.1.b**: There must exist a receive event $r'$ happens before $r$ in $Q^{i-1}$ such that $r' \in Cut(T, Q^{i-1})$. (Otherwise, $r \in Cut(T, Q^{i-1})$.) If $r'$ is in $RTR^{i-1}$, then $r$ cannot exist in $V^{i-1}$ since $r'$ will be reconciled in $V^{i-1}$, which will remove $r$ from $V^{i-1}$, leading to a contradiction. Otherwise, $r'$ will not be reconciled in $V^{i-1}$. Thus, $r' \in Cut(T, Q^i)$. This means that $r$ cannot exist in $Cut(T, Q^i)$, leading to a contradiction.

Next, we show $r \in RTR^{i-1}$. Assume otherwise. Then, there exists at least one receive event $r'$ in $Q^{i-1}$ such that $r'$ happens before $r$ in $Q^{i-1}$ and $send(r', Q^{i-1}) != send(r', T)$. Since $r \in Cut(T, Q^{i-1})$, $r'$ is not in $Cut(T, Q^{i-1})$. Therefore, $r'$ will not be reconciled at iteration $i - 1$. This implies that $r$ is not in $RTC^i$. Further, since each iteration introduces no new differences, if $r$ is not in $RTU^{i-1}$, then $r$ is not in $RTU^i$. It follows that r is not $RTR^i$, leading to a contradiction.

**Case 2.2**: Assume that *r.color = white* at iteration $j - 1$ and *r.color = black* at iteration $j$, $j < i$. Then, either (a) $send(r, V^j) != send(r, Q^j)$ or (b) there exists a receive event $r' \in RTR^j$ so that $send(r', V^j) != send(r', Q^j)$ and $r$ happens before $r'$ in $V^j$.

**Case 2.2.a**: In *Guided-RT*, the race outcome of $r$ can be changed only in (*). Thus, $send(r, V^j) = send(r, T)$. This means that $r$ is not in $Cut(T, Q^i)$, leading to a contradiction.

**Case 2.2.b**: Note that $r \in Diff(T, Q^j)$. Since $r$ happens before $r'$ in $V^j$, $r'$ cannot be in $Cut(T, Q^j)$, leading to a contradiction.

**Case 2.3**: Assume that *r.color = white* at iteration $j - 1$ and *r.color = gray* at iteration $j$, $j < i$. Then, there exists a receive event $r'$ in $RTR^j$ so that $send(r', V^j) != send(r', Q^j)$, and $r$ happens before $send(r', Q^j)$ in $V^j$ but is not in $p\text{-}struct(r', V^j)$.

First, we show that *r'* must be in $RTU^j$. It suffices to show that $send(r', Q^j)$ exists in both $V^j$ and $Q$, and thus $r$ also happens before $send(r', Q^j)$ in $Q^j$ (since the prime structure $send(r', Q^j)$ in $Q^j$ must be the same as that in $V^j$. Assume otherwise. There must exist a receive event $r''$ in $RTR^j$ such that $r'' \rightarrow_Q send(r', Q^j)$. This implies that $r'$ cannot be in $RTR^j$, leading to a contradiction.

Next we show that changing the send partner of $r$ to $send(r, T)$ reconciles the last difference of a tangled cycle. Assume that $r$ and *r'* are involved in a tangled cycle $C$. Since *r'* is in $RTR^j$, *r'* is reconciled at iteration $j$. This means that $C$ is broken at iteration $j$. Since r is in $Cut(Q, T)$ at iteration $i$, all the differences in $C$ must have been completed between iteration $j$ and iteration $i$. Therefore, changing the send partner of $r$ to $send(r, T)$ reconciles the last difference of $C$.

**Lemma 2**: Let $Q_1$ and $Q_2$ be two SR-sequences of program $P$ with input $X$. If $Cut(Q_1, Q_2)$ is empty, then $Q_1$ and $Q_2$ are the same SR-sequence.

**Proof**: This lemma can be easily established, considering that any difference between $Q_1$ and $Q_2$ can be traced back to a message race.

**Lemma 3**: Algorithm *Guided-RT* must terminate.

**Proof**: We show that *RTR* must be non-empty at each iteration. This implies that at least one difference between $Q$ and $T$ will be reconciled at each iteration. Since the number of differences between $Q$ and $T$ is finite, *Guided-RT* must terminate.

We proceed by contradiction. Assume that both *RTC* and *RTU* are empty at iteration $i$. Let $r_1 \in Cut(T, Q_i)$. There must exist a receive event $r_1' \in Diff(T, Q_i)$ such that $r_1'$ happens before $r_1$ in $Q_i$ and $r_1$ and $r_1'$ are not involved in a tangled cycle. Note that $r_1'$ must not be in $Min(Diff(T, Q_i))$, as otherwise, $r_1$ cannot be in $Cut(T, Q_i)$. Therefore, there exists a receive event $r_2 \in Cut(T, Q_i)$ such that $r_2 \rightarrow_T r_1'$. Note that $r_2 \neq r_1$. Similarly, there must exist a receive event $r_2' \in Diff(T, Q_i)$ such that $r_2'$ happens before $r_2$ in $Q_i$ and $r_2$ and $r_2'$ are not involved in a tangled cycle, and a receive event $r_3 \in Cut(T, Q_i)$ such that $r_3$ is different from $r_1$ and $r_2$. Assume that the number of receive events in $Cut(T, Q_i)$ is $n$. We can repeat the above procedure to derive a sequence of events $r_1, r_2, ..., r_n, r_{n+1}$, where $r_{n+1}$ must be different from $r_1, ..., r_n$, leading to a contradiction.

### 1.2 Part II of Theorem 2

We first introduce some definitions that are needed in our proof.

**Definition 1**: Let $Q_1$ and $Q_2$ be two SR-sequences exercised during reachability testing. $Q_1$ is the parent of $Q_2$ if $Q_2$ is an SR-sequence exercised by prefix-based testing with a variant $V$ of $Q_1$. In this case, it is also said that $V$ leads to $Q_2$. $Q_1$ is an ancestor of $Q_2$ if $Q_1$ is the parent of $Q_2$ or there exists an SR-sequence $Q_3$ such that $Q_1$ is the parent of $Q_3$ and $Q_3$ is an ancestor of $Q_2$.

**Definition 2**: Let $Q_1$ and $Q_2$ be two SR-sequences exercised during reachability testing. $Q_1$ and $Q_2$ are siblings if neither $Q_1$ is an ancestor of $Q_2$ nor $Q_2$ is an ancestor of $Q_1$.

**Definition 3**: Let $Q_1$ and $Q_2$ be two SR-sequences such that $Q_1$ is an ancestor of $Q_2$. Then, $between(Q_1, Q_2) = \{Q \mid Q_1$ is an ancestor of $Q$ *and* $Q$ is an ancestor of $Q_1\}$.

The main idea of our proof is to show that given two sequences $Q_1$ and $Q_2$ exercised by reachability testing, there exists a race difference between $Q_1$ and $Q_2$, in the following two cases: (1) $Q_1$ is an ancestor of $Q_2$ or $Q_2$ is an ancestor of $Q_1$; (2) $Q_1$ and $Q_2$ are siblings.

**Lemma 1**: Let $Q_1$ and $Q_2$ be two SR-sequences exercised by our reachability testing algorithm. There exists at least one race difference between $Q_1$ and $Q_2$ if $Q_1$ is an ancestor of $Q_2$ or $Q_2$ is an ancestor of $Q_1$.

**Proof**: By symmetry, it suffices to show that if $Q_1$ is an ancestor of $Q_2$, then there exists at least one race difference between $Q_1$ and $Q_2$.

Let $V$ be the variant of $Q_1$ that leads to $Q_2$. There must exist one race difference between $Q_1$ and $V$. Let $r$ be a receive event in $Q_1$ and $V$ so that $send(r, Q_1) \neq send(r, V)$. Therefore, the color of $r$ in $V$ is black. According to our algorithm, the race outcome of $r$ will never be changed in subsequent iterations. In addition, $\forall r'$: $r' \rightarrow_V r \Rightarrow$ the color of $r$ in $V$ is black. This means that we cannot change the race outcome of any receive happening before $r$ in $V$ in subsequent iterations either. Therefore, $r$ will never be removed. Hence, $r$ must exist in both $Q_1$ and $Q_2$ and $send(r, Q_1) \neq send(r, Q_2)$.

**Lemma 2:** Let $Q_1$ and $Q_2$ be two sibling sequences. Then, there must exist at least one race difference between $Q_1$ and $Q_2$.

**Proof**: Let $Q$ be the youngest common ancestor of $Q_1$ and $Q_2$. Let $V_1$ and $V_2$ be the two variants of $Q$ that lead to $Q_1$ and $Q_2$, respectively. There must exist at least one race difference between $V_1$ and $V_2$. Let $r$ be a receive event in $V_1$ and $V_2$ so that $send(r, V_1) \neq send(r, V_2)$. We consider the following cases: (1) $send(r, V_1) \neq send(r, Q)$, and $send(r, V2) \neq send(r, Q)$; (2) $send(r, V_1) = send(r, Q)$ (of course, $send(r, V_2) \neq send(r, Q)$.) (3) $send(r, V_2) = send(r, Q)$ (of course, $send(r, V_1) \neq send(r, Q)$.). By symmetry, we only need to consider cases (1) and (2).

**Case (1)**: According to our algorithm, the color of $r$ in $V_1$ and $V_2$ is black. This means that the race outcome of $r$ will never be changed in $V_1$ and $V_2$. In addition, the color of any receive event *r'* that happens before $r$ in $V_1$ and $V_2$ is also black. This means that the race outcome of any receive event happening before $r$ will not be changed in both $V_1$ and $V_2$. Therefore, $r$ will never be removed from $V_1$ and $V_2$. Hence, the race difference with $r$ will be preserved in $Q_1$ and $Q_2$.

**Case (2)**: Note that the color of $r$ in $V_2$ is black, and the color of $r$ in $V_1$ is either gray or white. Also note that since $send(r, V_2) \in race(r, Q)$, $send(r, V_2)$ is also in $V_1$.

Let $Q'$ be an arbitrary sequence in $between(Q, Q_1)$. Then, $send(r, V_2)$ is an *old* send event in $Q'$ (because a send event can never be recollected by Proposition 2). If $r$ is an *old* receive event in $Q'$, then $send(r, V_2)$ will be removed from $race(r, Q')$. As a result, we will not be able to change the send partner of $r$ in $Q_1$ to $send(r, V_2)$. Therefore, $send(r, Q_1) \neq send(r, Q_2)$. (Note that $send(r, V_2) = send(r, Q_2)$, because the color of $r$ in $V_2$ is black, which means its send event can never be changed afterwards.)

In the following, assume that $r$ was removed in a race variant $V'$ and then recollected in the SR-sequence $Q' \in between(Q, Q_1)$, which was collected from prefix-based testing with $V'$. Therefore, there exists a receive event $r'$ in $Q$ such that $r'$ happens before $r$, $r'$ is not in $p\text{-}struct(r, Q)$, and $send(r', Q) \neq send(r', Q')$. Then, if $send(r', Q_1) \neq send(r', Q_2)$, there exists a race difference between $Q_1$ and $Q_2$. Otherwise, we show that there must exist another race difference between $Q_1$ and $Q_2$.

Note that $send(r', Q) = send(r', V_1) = send(r', V_2)$, $send(r', V_1) \neq send(r', Q')$, and $send(r', Q') = send(r', Q_1)$. Thus, $send(r', V_2) \neq send(r', Q_1)$. By assumption, $send(r', Q_1) = send(r', Q_2)$. Thus, there must exist a sequence $Q'' \in between(V_2, Q_2)$ such that $send(r', V_2) != send(r', Q'') (= send(r', Q_2))$. Note that the color of $r'$ in $V_2$ is gray (since $send(r, Q) \neq send(r, V_2)$). Therefore, changing the send partner of $r'$ to $send(r', Q'')$ must reconcile the last difference of a tangled cycle $C$ w.r.t. $Q$ and $Q''$ (and thus $Q_2$) in which $r'$ is the non-cycle-breaking event of the smallest *weakly-reconciled-before* link. If there exists no race difference between $Q_1$ and $Q_2$, then $C$ is also a tangled cycle w.r.t. $Q$ and $Q_1$, and the last difference of $C$ must be completed at another receive event $r''$. This is impossible, because by assumption, there is no interconnected cycle and the only way to break a tangled cycle is to reconcile the difference with the smallest cycle-breaking event.