Stateless Techniques for Generating Global and Local Test Oracles for Message-Passing Concurrent Programs

R. Carver
rickhcarver@gmail.com

Yu Lei
Dept. of Computer Science and Engineering
University of Texas at Arlington,
Arlington, Texas 76019
ylei@cse.uta.edu

Abstract. A *test oracle* for a concurrent program is a method for checking whether an observed behavior of the program is consistent with the program's specification. Abstract specification models for message-passing concurrent programs are often expressed as, or can be translated into, a labeled transition system (LTS). Stateful techniques for generating test oracles from LTS specification models are often limited by the *state explosion problem*. In this paper, we present a *stateless* technique for generating global and local test oracles from LTS specification models. A *global test oracle* uses tests generated from a global LTS model of the complete system to verify a global implementation relation between the model of the system and its implementation. Global test oracles, however, may require too many test sequences to be executed by the implementation. A *local test oracle* verifies local implementation relations between individual component models and their implementation threads. Local tests are executed against individual threads, without testing the system as a whole. Verifying the local implementation relations implies that a corresponding global implementation relation holds between the complete system model and its implementation. Empirical results indicate that using local test oracles can significantly reduce the number of executed test sequences.

Keywords: concurrent programs; model-based testing; stateless search; test oracle; test sequences.

1. Introduction

A concurrent program contains two or more threads that communicate and synchronize with each other to perform some task. Because of their non-deterministic execution behavior, concurrent programs are notoriously difficult to test. A *test oracle* for a concurrent program is a method for checking whether an observed behavior of the program is consistent with the requirements or specification. *Model-based testing* uses abstract specification models both as a source of test cases and as a source of information for the test oracle.

Abstract specification models for concurrent programs are often expressed as, or can be translated into, a *labeled transition system* (LTS). An LTS models program behavior as a type of state machine. Each state in an LTS is an abstraction of a state in the program. Transitions are labeled with the abstract program events performed during state transitions.

Given a specification model M, a test case corresponds to a selected sequence of transitions in the LTS representing M. The transition labels of such a sequence define a *test sequence*. A test oracle can be constructed from M as follows: if M accepts a test sequence, then so should a concrete implementation CP of M. This *implementation relation* between M and CP is denoted M ≤ CP [Brinksma 1988; Brinksma and Scollo 1986; Brinksma and Scollo 1987; Pitt and Freestone 1990; Tretmans 1999].

Existing test oracles that are constructed by selecting test sequences from model M have a major limitation – while they can check whether the relation M ≤ CP holds, they cannot check whether CP ≤ M holds. Relation CP ≤ M says that if M does not accept a test sequence, then neither should CP.

A second limitation of existing test oracles arises from the potentially huge number of states in the LTS representing M. The number of states may be especially large when an *interleaving concurrency model* is used to construct the state space of M. In such a state space, a test sequence corresponds to a sequence of states, called an *interleaving sequence*, in which concurrent transitions are executed one at a time, in an arbitrary order. The state space contains one interleaving sequence for each possible order in which the concurrent transitions can be executed. The resulting explosion in the number of modeled states may prevent a complete state space of M from being built and/or make it impractical to execute the implementation with all of the possible test sequences that can be generated from M.

A different approach to constructing a test oracle is to build multiple, local test oracles. A *local test oracle* verifies a local implementation relation between each individual LTS component model in M and its corresponding implementation thread in CP. Local tests are executed against individual threads, without testing the system as a whole. The number of local tests generated for an individual thread is typically small. Thus, the total number of local tests may be significantly less than the number of global tests that can be generated from the complete model.

A local test oracle can be generated by analyzing a single LTS component model in M. However, some of the local test sequences that are generated from a single component may not be allowed in the global context of M, which may lead to inconclusive or incorrect test results. For example, a single LTS component may allow messages to be received in an order that is not possible if one also considers the constraints that the other components in M impose on the order in which messages can be sent. Thus, a failure that occurs when a local test is executed may not be possible when the system is executed.

Existing techniques for generating local oracles ensure that local oracles are consistent with the system by considering all of the LTS components in model M when generating a local oracle [Van der Bijl et al. 2004; Gotzhein and Khendek 2006; Carver and Lei 2013; Faivre et al. 2007; Kanso et al. 2012]. These techniques are limited by the potential for state explosion during oracle generation, or by the restrictions the techniques impose on the structure of model M such as each state must specify a response for every possible modeled input, or model M must contain only two components.

The salient contributions of this paper are:

- Test oracles for equivalence checking: We present test oracles for checking the stronger relation M ≈ CP, which holds when M and CP accept the same test sequences. Furthermore, we show that this equivalence check can be performed using the same test sequences that are generated from M to verify the weaker relation M ≤ CP. The test sequences contain additional information that makes it possible to compare the non-determinism that is present in M to that in CP, when each of them accepts a given test sequence. If CP is more non-deterministic than M, then CP can accept a test sequence that M cannot accept.
- Stateless oracle generation: Stateless techniques for generating global and local test oracles do not require any states of model M to be stored, and thus do not require the large amount of memory that may be used by stateful techniques. We have adapted that stateless search algorithm in [Lei and Carver 2006] to LTS models. This algorithm, is interleaving-free, and guarantees that two test sequences never differ only in the order of concurrent events. An embarrassingly parallel version of this algorithm can be executed on a cluster of workstations [Carver and Lei 2010a]. This is the first time we are aware of that a stateless, interleaving-free search algorithm has been applied to LTS models.
- Local test oracles: Local testing can be used to test implementation threads separately while still verifying that the global relation M ≈ CP holds between the complete system model and its implementation. Empirical results indicate that using local test oracles can significantly reduce the number of test sequences that must be executed against the implementation.
- Implementation: Our techniques for generating global and local test oracles have been implemented using the *Modern Multithreading* class library [Carver and Tai 2006]. This library contains thread and synchronization classes that provide testing and debugging services for multithreaded programs.

Fig. 1 shows our test automation framework for generating global and local oracles. Tests are generated from a model M that specifies a set of LTS component models and their parallel composition. Model M may be built by translating a model written in a high-level specification language into a set of

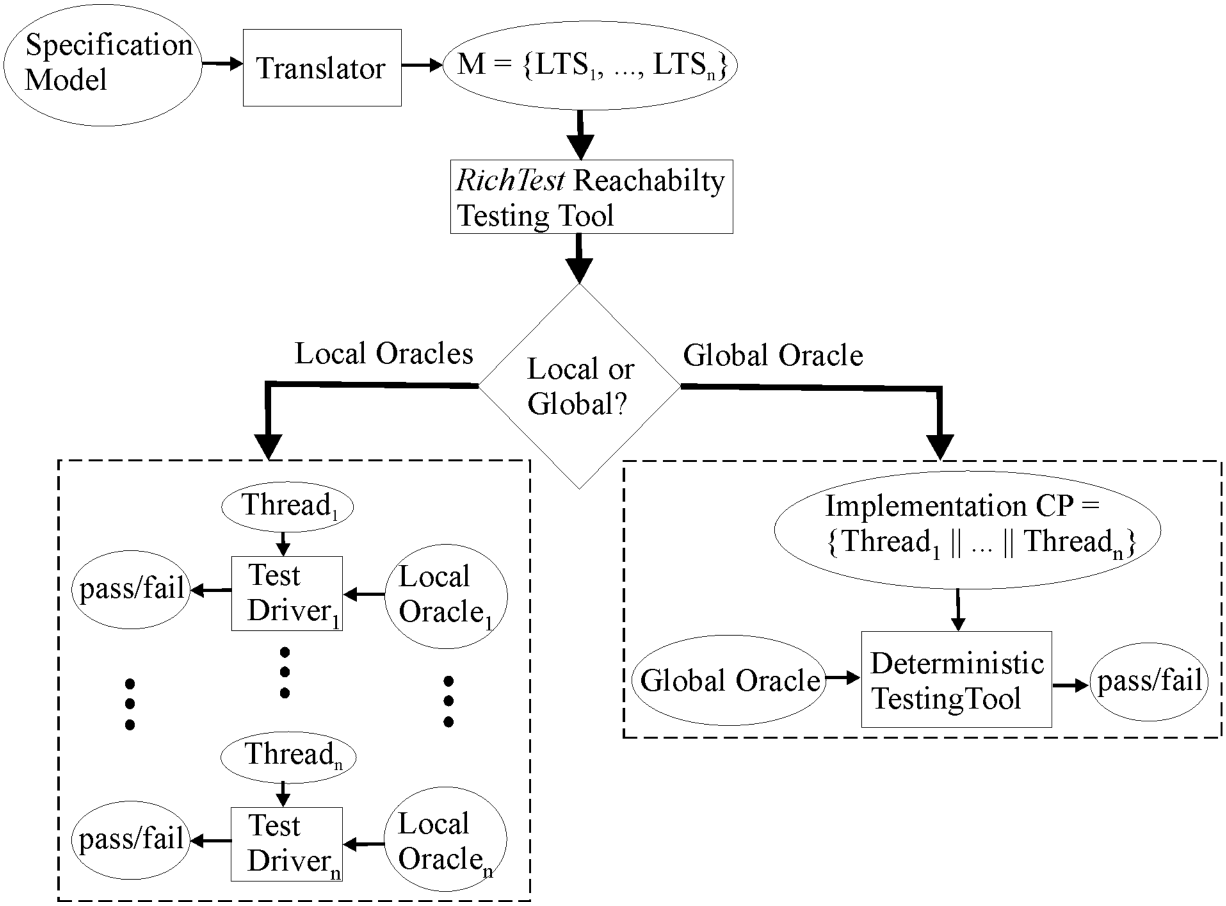


Figure 1. Test automation framework.

LTS components. We used specification models that were written in the Lotos language [Bolognesi and Brinksma 1987] and translated using CADP [Fernandez et al. 1996], in the empirical study reported in Section 9. The LTS component models of M are input into the *RichTest* reachability tool, which outputs either a global test oracle for implementation CP or a set of local oracles - one for each thread in CP. A deterministic testing tool is used to determine whether CP can exercise the thread interactions specified by the global test sequences. A test driver is used to determine whether the interactions specified by the local test sequences of a local oracle can be exercised by the corresponding implementation thread.

To illustrate these contributions and our testing framework, consider an LTS model and Java implementation of a well-known distributed mutual exclusion (DME) algorithm [Ricart and Agrawala 1981]. As reported in Section 9, attempts to build the complete state space of the DME model failed due to state explosion. Generating test sequences on-the-fly from the DME model [Tretmans 1999; Tretmans and Brinksma 2003] avoids state space construction, but does not reduce the number of interleaving-based sequences, which we report as more than 72 x 10^{33} sequences. The stateless, global test oracle presented in this paper consist of approximately 1.45 billion global test sequences generated from the

DME model, but this is still too many sequences to execute against the DME implementation. Attempts to generate local oracles from reduced state space models, as described in [Carver and Lei, 2013], failed due to intermediate state explosion. The local oracles presented in this paper use a total of 7,126 local test sequences extracted from the 1.45 billion global sequences generated from the DME model. Extracting these 7,126 local sequences from the 1.45 billion global sequences and executing the local sequences against the individual implementation threads takes considerably less time than generating the 1.45 billion global sequences and executing all 1.45 billion sequences against the complete implementation. Furthermore, the 7,126 local test sequences can be used to verify that the DME model and its implementation accept the same sequences.

There are several important assumptions associated with our stateless test generation algorithm. The first assumption is that the sole source of non-deterministic behavior in an LTS component or implementation thread is the non-deterministic order in which the component or thread receives messages. The second assumption is that the model is assumed to be closed, i.e., the environment with which the implementation interacts is modeled by LTS components in the model. These assumptions are discussed in detail in Section 6. Additional assumptions about the model are discussed in Section 2.

The remainder of this paper is organized as follows. Section 2 provides background information about LTS models and test sequences for concurrent programs. Section 3 describes the execution models that we use for stateless searches of LTSs. Section 4 formally defines the global implementation relations $M \leq CP$, $CP \leq M$, and $M \approx CP$, and their local counterparts. Section 5 presents an overview of the stateless search algorithm in [Lei and Carver 2006]. Sections 6 and 7 show how to use this stateless search algorithm to generate global test oracles for relations $M \leq CP$ and $M \approx CP$, respectively. Section 8 shows how to derive local test oracles for verifying relations $M \leq CP$ and $M \approx CP$. Section 9 reports the results of an empirical study on generating global and local oracles. Section 10 surveys related work. Section 11 provides concluding remarks and presents our plans for future work.

2. Background

The stateless test oracle presented in this paper is for concurrent programs that use message passing for communication and synchronization. The intended behavior of a message-passing program is modeled using an extended LTS model called an annotated LTS [Koppol et al. 2002]. A test sequence of a message-passing concurrent program is a sequence of send and receive events, called an SR-sequence. In this section, we provide definitions of LTS and ALTS models and SR-sequences.

2.1 Labeled Transition Systems (LTS)

LTS models contain nodes representing the states of a program and labeled edges representing transitions from state to state.

Definition 2.1: An LTS is a 4-tuple <Q, E, R, $q0$>, where Q is a non-empty finite set of states, E is a finite set of transition labels, $R \subseteq Q \times E \times Q$ is the transition relation, and $q0$ is a state in Q denoting the initial state. As in CCS [Milner 1989], we assume a set C of channel names exists, and we let the set of labels $E = C \cup \overline{C}$, where $\overline{C} = \{ \overline{a} \mid a \in C\}$.

For message-passing programs, which synchronize by sending and receiving messages, the labels in E encode send and receive events, where $\overline{e}$ represents a send event and e a receive event. An LTS may contain one or more *termination states*, which are states without outgoing transitions.

Fig. 2 shows an LTS with 4 states and 3 transitions. State 0 is the start state and states 2 and 3 are termination states. The transition labeled $\overline{p1}$ represents a send event, and transitions $p2$ and $p3$ represent receive events.

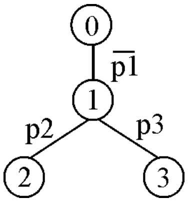


Figure 2. LTS model.

In this paper, we assume that each send or receive event in the implementation is expected to appear in the LTS model of the implementation. The model may, however, contain "extra" events that are not actually implemented. These events, e.g., can be used for specifying and verifying correctness properties of the model. We assume that the extra events are removed from the LTS models before test sequences are generated from the models.

We also assume that component models do not contain any unobservable events, called τ (tau) events. Tau events can be used to model design decisions that are yet to be made about implementation threads. However, we assume that high- and low-level design decisions about the communication and synchronization behavior of the components/threads have been made and are reflected in the models when testing starts. Our stateless search algorithm does not generate any tau events when test sequences are generated.

2.2 Annotated Labeled Transition Systems (ALTS)

The send and receive events in an LTS model are encoded by simple transition labels. Formats that have been developed for representing test sequences for implementations encode send and receive events with more complex event descriptors [Tai 1985, Tai et al. 1991, Tai and Carver 1996, Lei and Carver 2006]. The event descriptors contain implementation information such as the IDs of the sending and receiving threads, the operation performed, and the source port of the operation. A port p is a communication channel through which messages are sent using *p.send()* and received using *p.receive()*. Only one thread can receive messages from a given port; this thread is called the *owner* of the port.

Koppol et al. [2002] extended the LTS model and the algebraic laws used in CCS to allow implementation information, in the form of transition annotations, to be encoded in an LTS. Their extended LTS model is called an annotated labeled transition system (ALTS). As we will show, the annotations are used by the stateless search algorithm to generate test sequences. Annotations are also used to build an efficient partial-order representation of a test sequence, generate local test sequences, and transform an abstract test sequence of the model into a concrete test sequence of the implementation.

Definition 2.2: An ALTS is an LTS in which each transition is annotated with information about the corresponding implementation event. Formally, an ALTS is a 5-tuple $<Q, E, A, R, q0, N>$, where

- Q is a set of states
- E is a set of event labels
- A is a set of annotations
- $R \subseteq Q\times E(A)\times Q$ is the transition relation
- $q0$ is a state in Q denoting the initial state
- N is an annotation function defined next.

A transition $t\in R$ has an annotation $a\in A$ that is determined as follows:

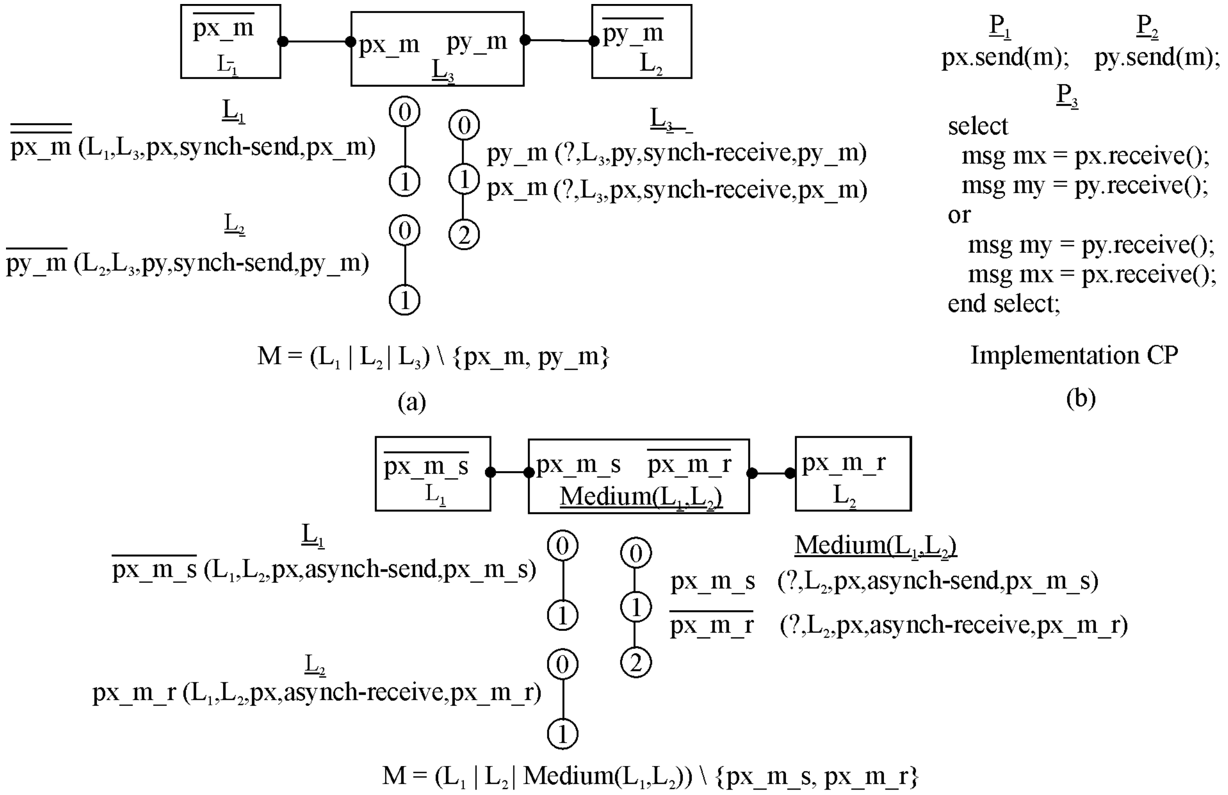
- A transition labeled with a send event $\bar{e} \in E$ is annotated as (L_i, L_j, p, op, e) if L_i is the ALTS executing this transition, L_j is the ALTS that receives the message, and p is the port that is accessed. The operation op is *synch-send* for synchronous message passing and *asynch-send* for asynchronous message passing. The label is e.

- A transition labeled with a receive event $e \in E$ is annotated as (*?*, L_j, *p*, *op*, *e*), if L_j is the ALTS executing this transition and *p* is the port that is accessed. The question mark symbol denotes that the identifier of the sending ALTS will be determined when this transition is involved in a synchronization with a matching send event (see Section 3). The operation *op* is *synch-receive* for synchronous message passing and *asynch-receive* for asynchronous message passing. The label is *e*.

For readability, a transition $(q, f(a), q') \in R$ with label *f* and annotation *a* is denoted as $q \xrightarrow{f(a)}_R q'$, with the transition annotation shown in parentheses next to the transition label. The transition can also be denoted without specifying label f (or $\overline{f}$), as in $q \xrightarrow{a}_R q'$, since the information in annotation *a* includes the label and the operation (*send* or *receive*) associated with the transition. The annotation of transition *t* is referred to as *t.annotation*. The individual fields of a transition annotation can also be referenced using dot notation. For example, the source *port* in an annotation for transition *t* is referred to as *t.port.* Techniques that have been developed for generating the annotations in an ALTS are described below.

The values for ALTSs L_i and L_j in an annotation are handled differently for synchronous and asynchronous message passing. When modeling synchronous message passing, the sending and receiving ALTSs are modeled as having a direct, synchronous interaction with each other, so the values of L_i and L_j in an annotation refer to the IDs of the LTSs in the interaction. Fig. 3(a) shows components L_1, L_2, and L_3 of a model M. The three boxes give static information about each of components L_1, L_2, and L_3. A box has a name that identifies the component and an interface that shows the ports that a component uses to communicate with other components. The ALTSs define the dynamic behavior of the components. Model M is defined as a composition of its components $(L_1 \mid L_2 \mid L_3) \setminus \{px\_m, py\_m\}$. The restriction operator $\setminus\{px\_m, py\_m\}$ denotes that only L_1, L_2, and L_3 can synchronize on *px_m* and *py_m*, i.e., they are not visible to components in the environment of M. Fig. 3(b) shows three implementation components P_1, P_2, and P_3 of an implementation CP that implements components L_1, L_2, and L_3 of model M. Components P_1 and P_2 execute *send* operations to P_3. Component P_3 executes a select statement with two alternatives. The two alternatives allow P_3 to receive the messages sent by P_1 and P_2 in either order.

Since LTS models use synchronous message passing semantics, asynchronous message passing must be simulated by using synchronous communication. This is done by having the sending and


px_m
L1
px_m py_m
L3
py_m
L2
L1
px_m (L1,L3,px,synch-send,px_m)
0
1
L2
py_m (L2,L3,py,synch-send,py_m)
0
1
L3
0
1
2
py_m (?,L3,py,synch-receive,py_m)
px_m (?,L3,px,synch-receive,px_m)
M = (L1 | L2 | L3) \ {px_m, py_m}
px_m_s
L1
px_m_s px_m_r
Medium(L1,L2)
px_m_r
L2
L1
px_m_s (L1,L2,px,asynch-send,px_m_s)
L2
px_m_r (L1,L2,px,asynch-receive,px_m_r)
Medium(L1,L2)
px_m_s (?,L2,px,asynch-send,px_m_s)
px_m_r (?,L2,px,asynch-receive,px_m_r)
M = (L1 | L2 | Medium(L1,L2)) \ {px_m_s, px_m_r}

(a)

```
P1              P2
px.send(m);   py.send(m);

       P3
select
  msg mx = px.receive();
  msg my = py.receive();
or
  msg my = py.receive();
  msg mx = px.receive();
end select;
```

Implementation CP

(b)

(c)

Figure 3. ALTS models.

receiving ALTSs interact with an ALTS that models a reliable communication medium between them. In this case, the values of L_i and L_j in an annotation refer to the IDs of the sending and receiving ALTSs, respectively, not the medium involved in the synchronous communication. Medium objects model implementation ports not threads. Fig. 3(c) shows ALTSs L_1 and L_2, and an ALTS Medium(L_1, L_2) that is used for sending messages from L_1 to L_2. ALTS L_1 and ALTS Medium(L_1, L_2) first have a synchronous synchronization on *px_m_s*, which is annotated as an *asynch-send* event on port *px*, and then Medium(L_1, L_2) and L_2 have a synchronous synchronization on *px_m_r*, which is annotated as an *asynch-receive* event on port *px*. The annotations for both events specify L_1 and L_2 as the sender and receiver, respectively.

In general, we will denote the ALTS component that models the medium between an ALTS L_s and an ALTS L_r as Medium(L_s, L_r). ALTS L_s and Medium(L_s, L_r) synchronize on event *e_s*, while Medium(L_s, L_r) and L_r synchronize on event *e_r*. The annotations map both events to the same implementation port *p*. We assume that an ALTS model of a medium correctly models the behavior of an implementation port, based on the guarantees that the communication sub-system or message-passing library provides to the programmer about the behavior of ports [Cypher and Leu 1994]. This includes properties such as message buffering and message ordering. For example, Medium(L_1, L_2) in Fig. 3(c) ensures that a message cannot be received by L_2 before it is sent by L_1, and that FIFO communication is used to pass messages from L_1 to L_2.

The events in an ALTS model M may represent message-passing between threads, or they may instead represent I/O operations between the threads and their environment, e.g., reading from a keyboard, input sensor, or file. Model M thus identifies the inputs that are used to verify the implementation relation between M and implementation CP, and also the expected outputs of CP when CP is executed with the inputs in M. In the remainder of this paper, we will refer to the I/O values specified by M as the inputs and outputs of M, and the possible inputs and expected outputs, respectively, of CP. Likewise, if a sequence *s* of model M contains events that represent input and output operations, then the values specified in the input and output events of *s* are referred to collectively as the inputs and outputs of sequence *s*, denoted *inputs(s)* and *output(s)*, respectively. We assume that when sequence *s* is used to test CP, the inputs of *s* are translated into the required input format for CP. An input *X* of implementation CP consists of a sequence of input values for each thread.

Our test oracles do not require an ALTS model and its implementation to be written at the same level of abstraction. Models can be written in specification languages that are problem oriented, not implementation oriented [Zave 1984], and in specification styles [Vissers et al. 1988] that express solutions using implementation-independent structures. The specification languages need not provide all of the implementation details that are provided by programming languages, the latter being concerned with issues like efficient execution and opportune code reuse through inheritance.

Annotation information must be generated in order to create an ALTS model instead of an LTS model. The annotation information supplies the implementation details that are needed for test execution. Chen and Carver [1996] showed how annotation information can be incorporated into models that are written in the Lotos specification language. Transition annotations are automatically computed when the Lotos specification is translated into an LTS model. The value passing and matching semantics of Lotos ensure that synchronizing events have annotations with matching receiver IDs, ports, and labels. The annotation information appears as part of structured transition labels in the resulting LTS. This is analogous to the way event descriptors are generated when an implementation is executed. We used this approach in the empirical study in Section 9 to create ALTS models from Lotos specifications. Refer to Chen and Carver [1996] for details about this approach.

2.3 SR-sequences of the Implementation

A test sequence of a message-passing concurrent program is a sequence of synchronized send and receive events, called an SR-sequence. A send or receive event refers to the execution of a send or receive statement, respectively.

Let CP be an implementation with concurrent threads {P_1, P_2, …, P_n}. SR-sequences of CP can be totally or partially ordered.

2.3.1 Totally-Ordered SR-sequences of the Implementation

Information about each synchronization event in a totally-ordered SR-sequence of implementation CP is encoded using event annotations, which are analogous to transition annotations.

Definition 2.3: The *event annotation* for a synchronization event in a totally-ordered SR-sequence of implementation CP is *(sending thread, receiving thread, port, op, label)*, where *sending thread* is the thread executing the send event; *receiving thread* is the thread that receives the message; *port* is the source port, which is owned by the *receiving thread*; *label* is a string that encodes the event as described below; and *op* is *asynch-send* for an asynchronous send event, *asynch-receive* for an asynchronous receive event, and *synchronous-synchronization* for a synchronization between synchronous send and receive events.

Definition 2.4: A totally-ordered SR-sequence of implementation CP is a sequence of send and receive events ((*sending thread*$_1$, *receiving thread*$_1$, *port*$_1$, *op*$_1$, *label*$_1$), (*sending thread*$_2$, *receiving thread*$_2$, *port*$_2$, *op*$_2$, *label*$_2$), ...), where (*sending thread*$_i$, *receiving thread*$_i$, *port*$_i$, *op*$_i$, *label*$_i$) denotes the i^{th}, i>0, event in the SR-sequence.

A totally-ordered SR-sequence may or may not be allowed by the implementation.

Definition 2.5: A totally-ordered SR-sequence Q is *feasible* for implementation CP with input X if Q can be exercised during an execution of CP with input X. (The implementation reads input X from e.g., the keyboard, or a file.)

Example 2.1: Two feasible, totally-ordered SR-sequences of implementation CP in Fig. 3(b) are:

- (L_2, L_3, *py*, *synchronous-synchronization*, *py_m*).(L_1, L_3, *px*, *synchronous-synchronization*, *px_m*), and
- (L_1, L_3, *px*, *synchronous-synchronization*, *px_m*).(L_2, L_3, *py*, *synchronous-synchronization*, *py_m*).

Tools and techniques for determining the feasibility of a totally-ordered SR-sequence for implementation CP with input X are described in Section 3.3.

2.3.2 Partially-Ordered SR-sequences of the Implementation

The test generation technique described in Sections 5 and 6 models the execution of implementation CP as a *partially-ordered* SR-sequence.

Definition 2.6: An event annotation for a send or receive event in a partially-ordered SR-sequence of implementation CP has the format *(sending thread, receiving thread, port, op, label, j, i)*, where *sending thread* is the thread that sent the message; *receiving thread* is the thread that receives the message; *port* is the source port, which is owned by the *receiving thread*; *label* is a string that encodes the event as described below; *i* and *j* are event indices; and *op* is the operation performed.

- For a send event *s*, *op* is either *asynch-send* or a *synch-send* operation; *i* is the index of send event *s*, indicating that *s* is the *i*th event executed by the *sending thread*; and *j* is the index of the receive event *r* that is synchronized with send event *s*, indicating that *r* is the *jth* event executed by the *receiving thread*.
- For a receive event *r*, *op* is either *asynch-receive* or *synch-receive*; *i* is the index of receive event *r*, indicating that *r* is the *i*th event executed by the *receiving thread*; and *j* is the index of the send event *s* that is synchronized with receive event *r,* indicating that *s* is the *j*th event executed by the *sending thread*.

Definition 2.7: A *local sequence* of thread P_i in CP is a totally-ordered sequence of send and receive events (*(sending thread$_1$, receiving thread$_1$, port$_1$, op$_1$, label$_1$, j_1, 1), (sending thread$_2$, receiving thread$_2$, port$_2$, op$_2$, label$_2$, j_2, 2), ...)* executed by P_i, where *(sending thread$_k$, receiving thread$_k$, port$_k$, op$_k$, label$_k$, j_k, k)* denotes the *kth*, *k*>0, event in the local sequence. A send (receive) event in a local sequence of P_i has P_i as the *sending* (*receiving*) *thread*.

Definition 2.8 A local sequence s_{Pi} of thread P_i is *feasible* for P_i with input X if s_{Pi} can be exercised during an execution of P_i with input X.

A local-testing technique for determining the feasibility of a local sequence for P_i with input X is described in Section 8.2.

Definition 2.9: A partially-ordered SR-sequence *Q* of implementation CP is defined as a tuple $(s_{P1}, s_{P2}, ..., s_{Pi})$, where s_{Pi}, $0 < i \leq n$, is a local sequence of Thread P_i.

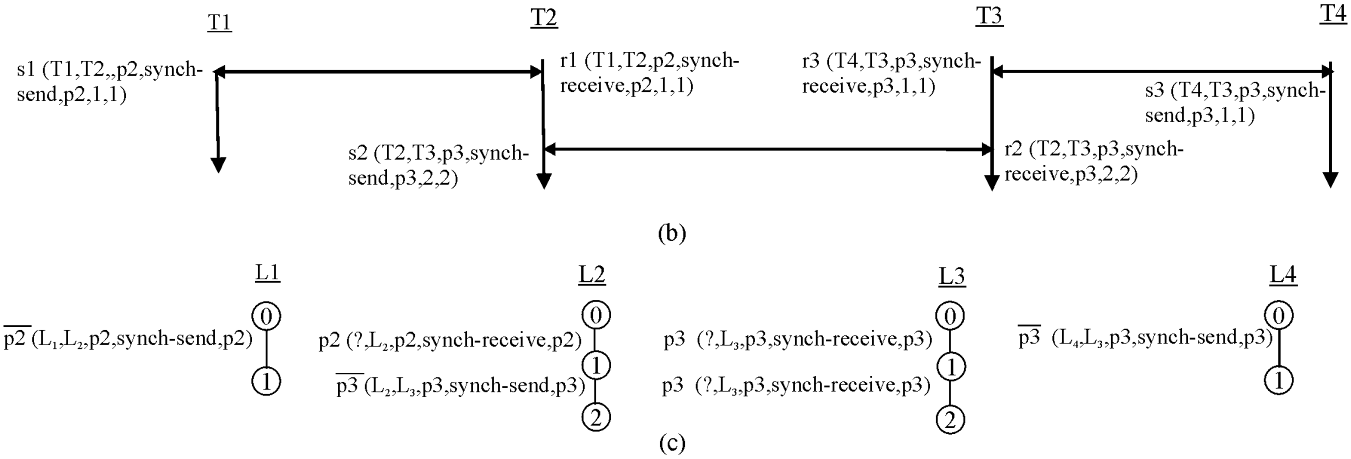
Definition 2.10: A partially-ordered SR-sequence *Q* is *feasible* for implementation CP with input X if *Q* can be exercised during an execution of CP with input X.

Tools and techniques for determining the feasibility of a partially-ordered SR-sequence for implementation CP with input X are described in Section 3.3.

A feasible partially-ordered SR-sequence can be depicted using a space-time diagram. Fig. 4(b) shows an SR-sequence of the synchronous message passing program in Fig. 4(a). The events are named, e.g., *s1* and *r1*, and the event annotations appear between parentheses next to the names. The local sequence for a thread is simply the send and receive events in the vertical time-line for the thread.

```
    T1              T2               T3              T4
p2.send();    p2.receive();    p3.receive();    p3.send()
              p3.send();       p3.receive;
```

(a)


T1
T2
T3
T4
s1 (T1,T2,,p2,synch-send,p2,1,1)
r1 (T1,T2,p2,synch-receive,p2,1,1)
r3 (T4,T3,p3,synch-receive,p3,1,1)
s3 (T4,T3,p3,synch-send,p3,1,1)
s2 (T2,T3,p3,synch-send,p3,2,2)
r2 (T2,T3,p3,synch-receive,p3,2,2)
L1
L2
L3
L4
0
1
2
$\overline{p2}$ (L₁,L₂,p2,synch-send,p2)
p2 (?,L₂,p2,synch-receive,p2)
$\overline{p3}$ (L₂,L₃,p3,synch-send,p3)
p3 (?,L₃,p3,synch-receive,p3)
p3 (?,L₃,p3,synch-receive,p3)
$\overline{p3}$ (L₄,L₃,p3,synch-send,p3)

(b)

(c)

Figure 4. Partially-ordered SR-sequence.

If a send event *s* is synchronized with a receive event *r* in a feasible SR-sequence, we refer to <*s, r*> as a *synchronization pair* and say that *s* is the send partner of *r* and *r* is the receive partner of *s*. Let *SP* be the set of synchronization pairs in an SR-sequence. The synchronization pairs in a feasible SR-sequence can be derived from the event indices of the send and receive events in the sequence. A receive event *(sending thread, receiving thread, port, op, label*, *j*, *i)* indicates that the j^{th} event of the *sending thread* and the i^{th} event of the *receiving thread* are a synchronization pair.

The synchronization pairs in a feasible SR-sequence must satisfy certain properties if program executions use a message-passing subsystem that has been correctly implemented by the operating system or run-time system [Cypher and Leu 1994]:

SP1: $<s,r> \in SP \Rightarrow s.i = r.j \wedge r.i = s.j$

SP2: $<s,r> \in SP \wedge <s,t> \in SP \Rightarrow r = t$

SP3: $<s,r> \in SP \wedge <t,r> \in SP \Rightarrow s = t$

Property SP1 ensures that the event indices of the send and receive events in a synchronization pair are consistent with each other. Properties SP2 and SP3 ensure that each event can be paired with at most one other event. We assume that the message-passing subsystem in the operating system or run-time system has been correctly implemented and thus that these properties hold in every feasible SR-sequence of an implementation.

A solid arrow in a space-time diagram drawn between a send event labeled *s* and a receive event labeled *r* indicates that *s* and *r* are a synchronization pair <*s*,*r*>. A double-headed arrow represents a message passed synchronously from *s* to *r.* A single-headed arrow represents an asynchronous message from *s* to *r.* We will use *send(r, Q)* to denote the send partner of receive event *r,* if the send partner exists, in an SR-sequence *Q*. Note that *send(r, Q)* is undefined if *r* is not synchronized with any send event in *Q*.

3. SR-sequences of the Model

Test oracles are constructed during a stateless search by composing component models to derive an SR-sequence that represents a single path through the complete system model. In this section, we define totally- and partially-ordered SR-sequences of the model. The format of an SR-sequence of the model is the same as that of an SR-sequence of the implementation. This makes the source of an SR-sequence, be it a model or an implementation, transparent to the stateless search algorithm.

The implementation relations and the test generation techniques presented later consider the SR-sequences of a model M that is comprised of a set of ALTSs $\{L_1, L_2, \ldots, L_n\}$. To generate SR-sequences of M, the component ALTSs in M are composed, but the result is an SR-sequence of M, not an intermediate or global composite ALTS. Recall that there are no tau transitions in any component ALTS, and that we model closed systems, i.e., the environment is modeled as an ALTS. Thus, an SR-sequence represents a sequence of synchronizations events between the component ALTS models of M, which includes M's environment.

3.1 Totally-Ordered SR-sequences of the Model

Information about each event in a totally-ordered SR-sequence of model M is encoded using event annotations.

Definition 3.1: The annotation for an event in a totally-ordered SR-sequence of model M is *(sending ALTS, receiving ALTS, port, op, label)*, where *sending ALTS* is the *ALTS* executing the send event; *receiving ALTS* is the *ALTS* that receives the message; *port* is the source port, which is owned by the *receiving ALTS*; *label* is a string that encodes the event as described below; and *op* is *asynch-send* for an asynchronous send event, *asynch-receive* for an asynchronous receive event, and *synchronous-synchronization* for a synchronization between a synchronous send and receive event.

These send and receive event annotations are the same as those used for totally-ordered SR-sequences of the implementation.

Definition 3.2: A totally-ordered SR-sequence of model M is a sequence of send and receive events ((*sending* $ALTS_1$*, receiving* $ALTS_1$*,* $port_1$*,* op_1*,* $label_1$*), (sending* $ALTS_2$*, receiving* $ALTS_2$*,* $port_2$*,* op_2*,* $label_2$*), ...*), where *(sending* $ALTS_i$*, receiving* $ALTS_i$*,* $port_i$*,* op_i*,* $label_i$*)* denotes the i^{th}, i>0, event in the SR-sequence.

A send or receive event occurs when a send transition of one component synchronizes with a compatible receive transition of another component. If the synchronization involves a *synch-send* by ALTS L_a and a *synch-receive* by ALTS L_b, then L_a and L_b synchronize directly with each other. Recall from Section 2.2 that if L_a executes an *asynch-send* to send a message to L_b, or if L_b executes an *asynch-receive* to receive a message sent by L_a, then ALTS Medium(L_a, L_b) is involved. Component L_a first synchronizes with Medium(L_a, L_b) on the *asynch-send* event and then Medium(L_a, L_b) synchronizes with

L_b on the *asynch-receive* event. This is modeled by having L_a and Medium(L_a, L_b) synchronize on an event labeled *e_s*, and by having Medium(L_a, L_b) and L_b synchronize on an event labeled *e_r*.

An *is-compatible-with* relation is defined between send and receive transitions of component ALTSs if they have compatible annotations. Compatible send and receive transitions can be synchronized.

Definition 3.3: $\mathit{Compatible}_{synch}\,(x_{source} \xrightarrow{\overline{e}(ax)}_{R_a} x_{dest}, y_{source} \xrightarrow{e(ay)}_{R_b} y_{dest})$ iff

$\mathrm{ax} = (L_a, L_b, p, \mathit{synch\text{ - }send}, e) \wedge \mathrm{ay} = (?, L_b, p, \mathit{synch\text{-}receive}, \mathrm{e})$.

Definition 3.4: $\mathit{Compatible}_{asynch\text{-}send}\,(\mathrm{x}_{\mathrm{source}} \xrightarrow{\overline{\mathrm{e\_s}}(\mathrm{ax})}_{\mathrm{R_a}} \mathrm{x}_{\mathrm{dest}}, \mathrm{y}_{\mathrm{source}} \xrightarrow{\mathrm{e\_s}(\mathrm{ay})}_{\mathrm{R}_{\mathrm{Mediun}(\mathrm{L_a}, \mathrm{L_b})}} \mathrm{y}_{\mathrm{dest}})$ iff

$\mathrm{ax} = (\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \mathrm{asynch\text{ - }send}, \mathrm{e\_s}) \wedge \mathrm{ay} = (?, \mathrm{L_b}, \mathrm{p}, \mathrm{asynch\text{ -}send}, \mathrm{e\_s})$.

Definition 3.5: $\mathit{Compatible}_{asynch\text{-}receive}\,(\mathrm{x}_{\mathrm{source}} \xrightarrow{\overline{\mathrm{e\_r}}(\mathrm{ax})}_{\mathrm{R}_{\mathrm{Mediun}(\mathrm{L_a}, \mathrm{L_b})}} \mathrm{x}_{\mathrm{dest}}, \mathrm{y}_{\mathrm{source}} \xrightarrow{\mathrm{e\_r}(\mathrm{ay})}_{\mathrm{Rb}} \mathrm{y}_{\mathrm{dest}})$ iff

$\mathrm{ax} = (\mathrm{L_a}, \mathrm{L_{b,}}\ \mathrm{p}, \mathrm{asynch\text{-}receive}, \mathrm{e\_r}) \wedge \mathrm{ay} = (?, \mathrm{L_b}, \mathrm{p}, \mathrm{asynch\text{-}receive}, \mathrm{e\_r})$.

Def. 3.3 defines a compatible, synchronous synchronization between sender L_a and receiver L_b. The usual synchronization rule for CCS is used, which requires matching event labels $\overline{e}$ and e for the synchronized pair of send and receive transitions $\mathrm{x}_{\mathrm{source}} \xrightarrow{\overline{\mathrm{e}}(\mathrm{ax})}_{\mathrm{R_a}} \mathrm{x}_{\mathrm{dest}}$ and $\mathrm{y}_{\mathrm{source}} \xrightarrow{\mathrm{e}(\mathrm{ay})}_{\mathrm{R_b}} \mathrm{y}_{\mathrm{dest}}$. This rule is extended to the annotations by also requiring annotations *ax* and *ay* to have a matching receiver L_b and a matching port *p*. This reflects the message-passing semantics of the implementation, which requires the sender and receiver to access the same port, for a port that is owned by the receiver. The operation types for *ax* and *ay* are required to be *synch-send* and *synch-receive*, respectively.

Defs. 3.4 and 3.5 represent an asynchronous send and an asynchronous receive, respectively. ALTS Medium(L_a, L_b) synchronizes with L_a on the *asynch-send* and with L_b on the *asynch-receive*. The event labels must match, as usual, and annotations *ax* and *ay* must have a matching receiver L_b and a matching port *p*. The operation types for annotations *ax* and *ay* are both required to be either *asynch-send* or *asynch-receive*.

Example 3.1: In Fig 3(c), ALTS *L1* synchronizes with Medium(*L1*, *L2*) on compatible transitions $(0 \xrightarrow{\overline{px\_m\_s}\ (L1,L2,px,asynch\text{-}send,px\_m\_s)}_{\mathrm{R_1}} 1$ and $0 \xrightarrow{px\_m\_s(?,L2,px,asynch\text{-}send,px\_m\_s)}_{\mathrm{R}_{\mathrm{Medium}(\mathrm{L_1}, \mathrm{L_2})}} 1)$. The event labels $\overline{px\_m\_s}$ and *px_m_s* match, the annotations have a matching receiver L_2, a matching port *px*, and matching operation type *asynch-send*.

A function *Ann* is defined for the annotation that represents a synchronization between two compatible send and receive transitions from two local source states to two local destination states. This annotation captures the IDs of the sending and receiving components that were synchronized.

Definition 3.6:

$\text{Ann}_{\mathit{synch}}(\text{x}_{\text{source}} \xrightarrow{\overline{\text{e}}(\text{L}_{\text{a}}, \text{L}_{\text{b}}, \text{p}, \text{synch-send}, \text{e})}_{\text{R}_{\text{a}}} \text{x}_{\text{dest}}, \text{y}_{\text{source}} \xrightarrow{\text{e}(?, \text{L}_{\text{b}}, \text{p}, \text{synch-receive}, \text{e})}_{\text{R}_{\text{b}}} \text{y}_{\text{dest}}) \equiv_{\text{def}}$

$(L_a, L_b, p, \mathit{synchronous\text{-}synchronization}, e)$.

Definition 3.7:

$\mathit{Ann}_{\mathit{asynch\text{-}send}}(\text{x}_{\text{source}} \xrightarrow{\overline{\text{e\_s}}(\text{L}_{\text{a}}, \text{L}_{\text{b}}, \text{p}, \text{asynch-send}, \text{e\_s})}_{\text{R}_{\text{a}}} \text{x}_{\text{dest}}, \text{y}_{\text{source}} \xrightarrow{\text{e\_s}(?, \text{L}_{\text{b}}, \text{p}, \text{asynch-send}, \text{e\_s})}_{\text{R}_{\text{Mediun(La,Lb)}}} \text{y}_{\text{dest}}) \equiv_{\text{def}}$

$(L_a, L_b, p, \mathit{asynch\text{-}send}, \mathit{e\_s})$.

Definition 3.8:

$\text{Ann}_{\mathit{asynch\text{-}receive}}(\text{x}_{\text{source}} \xrightarrow{\overline{\text{e\_r}}(\text{L}_{\text{a}}, \text{L}_{\text{b}}, \text{p}, \text{asynch-receive}, \text{e\_s})}_{\text{R}_{\text{Mediun(La,Lb)}}} \text{x}_{\text{dest}}, \text{y}_{\text{source}} \xrightarrow{\text{e\_r}(?, \text{L}_{\text{b}}, \text{p}, \text{asynch-receive}, \text{e\_r})}_{\text{R}_{\text{b}}} \text{y}_{\text{dest}})$

$\equiv_{\text{def}} (L_a, L_b, p, \mathit{asynch\text{-}receive}, \mathit{e\_r})$.

Def. 3.6 defines the annotation for a synchronous synchronization event that involves components L_a and L_b. The annotation has operation type *synchronous-synchronization*. The ID of the sending ALTS L_{a} in the annotation of the send transition is used as the sender ID in the annotation for the synchronous synchronization (L_a, L_b, p, *synchronous-synchronization*, e).

Defs. 3.7 and 3.8 define the annotation for an asynchronous synchronization event that involves either a sending ALTS L_a or a receiving ALTS L_b, and an ALTS Medium(L_a, L_b) that models the medium used to send messages from L_a to L_b. In Def. 3.7, the sending ALTS L_a executing transition $x_{source} \xrightarrow{\overline{e\_s}(\text{L}_{\text{a}}, \text{L}_{\text{b}}, \text{p}, \text{asynch-send}, \text{e\_s})}_{R_a} x_{dest}$ synchronizes with a receive transition $\text{y}_{\text{source}} \xrightarrow{\text{e\_s}(?, \text{L}_{\text{b}}, \text{p}, \text{asynch-send}, \text{e\_s})}_{\text{R}_{\text{Medium}(\text{L}_{\text{a}}, \text{L}_{\text{b}})}} \text{y}_{\text{dest}}$ of Medium(L_a, L_b). Both of these events have operation type *asynch-send* in their annotations. In the annotation that represents this synchronization, the event type is *asynch-send*, and the ID of ALTS L_b, not the ID of the medium, is used as the receiving ALTS. In Def. 3.8, ALTS Medium(L_a, L_b) executes a send transition $x_{source} \xrightarrow{\overline{e\_r}(\text{L}_{\text{a}}, \text{L}_{\text{b}}, \text{p}, \text{asynch-receive}, \text{e\_r})}_{R_{\text{Medium}(\text{L}_{\text{a}}, \text{L}_{\text{b}})}} x_{dest}$ that synchronizes with a receive transition $y_{source} \xrightarrow{e\_r(?, \text{L}_{\text{b}}, \text{p}, \text{asynch-receive}, \text{e\_r})}_{R_b} y_{dest}$ of the receiving ALTS L_b. Both of these events have operation type *asynch-receive* in their annotations. In the annotation that represents this synchronization, the event type is *asynch-receive*, and the ID of ALTS L_a, not the ID of the medium, is used as the sending ALTS. The ID of ALTS Medium(L_a, L_b) is not used in any annotations, as it is not mapped to an actual thread in the implementation; rather, ALTS Medium(L_a, L_b) specifies the behavior of implementation port *p*.

Example 3.2: Continuing Example 3.1, the annotation for the synchronization between L_1 and Medium(L_1, L_2) is (L_1, L_2, *px*, *asynch-send*, *px_m_1*). The ID of the receiving ALTS is L_2, indicating that this event models an asynchronous send event from L_1 to L_2.

Recall from Section 2.2 that component ALTS L_i of model M is the 5-tuple $<Q_i, E_i, A_i, R_i, q0_i, N_i>$, $1 \le i \le n$.

Definition 3.9: A global state *g* of model M is a tuple $<q^1, q^2, \ldots, q^n>$ where local state $q^i \in Q_i$ for every $i \in \{1..n\}$. The initial global state *g0* of M is the tuple of initial local states $<q0^1, q0^2, \ldots, q0^n>$.

A *synchronized step* from global state *g* to global state *g′* in model M is a synchronization between two compatible local transitions of ALTS components. A synchronized step modifies the local states of the synchronizing components and leaves the states of the other components unchanged.

Definition 3.10: Let L_a, L_b, and Medium(L_a, L_b) be component ALTSs of M. A *synchronized step* of M is a tuple of the form:

(a) $<g, (\mathrm{x}_{\mathrm{source}} \xrightarrow{\bar{\mathrm{e}}(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \mathrm{synch\text{-}send}, \mathrm{e})}_{\mathrm{R_a}} \mathrm{x}_{\mathrm{dest}}, \mathrm{y}_{\mathrm{source}} \xrightarrow{\mathrm{e}(?, \mathrm{L_b}, \mathrm{p}, \mathrm{synch\text{-}receive}, \mathrm{e})}_{\mathrm{R_b}} \mathrm{y}_{\mathrm{dest}})), g'>$, or

(b) $<g, (\mathrm{x}_{\mathrm{source}} \xrightarrow{\overline{\mathrm{e\_s}}(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \mathrm{asynch\text{-}send}, \mathrm{e\_s})}_{\mathrm{R_a}} \mathrm{x}_{\mathrm{dest}}, \mathrm{y}_{\mathrm{source}} \xrightarrow{\mathrm{e\_s}(?, \mathrm{L_b}, \mathrm{p}, \mathrm{asynch\text{-}send}, \mathrm{e\_s})}_{\mathrm{R_{Mediun(La,Lb)}}} \mathrm{y}_{\mathrm{dest}}), g'>$, or

(c) $<g, (x_{source} \xrightarrow{\overline{e\_r}(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \mathrm{asynch\text{-}receive}, \mathrm{e\_r})}_{R_{Mediun(La,Lb)}} x_{dest}, y_{source} \xrightarrow{e\_r(?, \mathrm{L_b}, \mathrm{p}, \mathrm{asynch\text{-}receive}, \mathrm{e\_r})}_{Rb} y_{dest}), g'>$,

where global state $g = <q^1, q^2, \ldots, q^n>$, global state $g' = <q^{1'}, q^{2'}, \ldots, q^{n'}>$, and the following conditions are satisfied for every $i \in \{1..n\}$:

For a step of form (a):

(1) $q^a = x_{source}$, $q^{a'} = x_{dest}$, $q^b = y_{source}$, and $q^{b'} = y_{dest}$

(2) if $i \neq a$ and $i \neq b$ then $q^i = q^{i'}$

(3) Compatible($x_{source} \xrightarrow{\bar{e}(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \mathrm{synch\text{-}send}, \mathrm{e})}_{R_a} x_{dest}, y_{source} \xrightarrow{e(?, \mathrm{L_b}, \mathrm{p}, \mathrm{synch\text{-}receive}, \mathrm{e})}_{R_b} y_{dest}$).

For a step of form (b):

(1) $q^{a} = x_{source}$, $q^{a'} = \mathrm{x}_{\mathrm{dest}}$, $q^{\mathrm{Medium}(La,\, Lb)} = y_{source}$, and $q^{\mathrm{Medium}(La,\, Lb)'} = y_{dest}$

(2) if $i \neq a$ and $i \neq \mathrm{Medium}(L_a, L_b)$ then $q^{i} = q^{i'}$

(3) Compatible($x_{source} \xrightarrow{\overline{e\_s}\,(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \text{asynch-send}, \mathrm{e\_s})}_{R_a} x_{dest}, y_{source} \xrightarrow{e\_s\,(?, \mathrm{L_b}, \mathrm{p}, \text{asynch-send}, \mathrm{e\_s})}_{R_{Mediun(La,Lb)}} y_{dest}$)

For a step of form (c):

(1) $q^{\mathrm{Medium}(La,\, Lb)} = x_{source}$, $q^{\mathrm{Medium}(La,\, Lb)'} = x_{dest}$, $q^{b} = y_{source}$, and $q^{b'} = y_{dest}$

(2) if $i \neq \mathrm{Medium}(L_a, L_b)$ and $i \neq b$ then $q^{i} = q^{i'}$

(3) Compatible($x_{source} \xrightarrow{\overline{e\_r}(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \text{asynch-receive}, \mathrm{e\_r})}_{R_{Mediun(La,Lb)}} x_{dest}, y_{source} \xrightarrow{e\_r(?, \mathrm{L_b}, \mathrm{p}, \text{asynch-receive}, \mathrm{e\_r})}_{Rb} y_{dest}$).

A synchronized step from global state *g* to global state *g′* involves two transitions, which are either (a) synchronous send and receive transitions from L_a and L_b; (b) send and receive transitions from L_a and Medium(L_a, L_b) that model an asynchronous send; or (c) send and receive transitions from Medium(L_a, L_b) and L_b that model an asynchronous receive. The local states of the synchronizing components are modified (1) and the local states of the other components are unchanged (2). The synchronized transitions must be compatible (3).

A feasible SR-sequence of model M is a sequence of synchronized steps between the components in M.

Definition 3.11: A totally-ordered SR-sequence ($(Ls_1, Lr_1, p_1, op_1, e_1), (Ls_2, Lr_2, p_2, op_2, e_2), \ldots, (Ls_k, Lr_k, p_k, op_k, e_k)$) is *feasible* for model M if there is a sequence $g_1, g_2, \ldots, g_k$ of global states such that for every $i \in \{1 \,..\, k-1\}$:

$<g_i$, ($\mathrm{x_{source}} \xrightarrow{\bar{\mathrm{e}}(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \text{synch-send}, \mathrm{e})}_{\mathrm{R_a}} \mathrm{x_{dest}}, \mathrm{y_{source}} \xrightarrow{\mathrm{e}(?, \mathrm{L_b}, \mathrm{p}, \text{synch-receive}, \mathrm{e})}_{\mathrm{R_b}} \mathrm{y_{dest}}$) ,$g_{i+1}>$, or

$<g_i$,($\mathrm{x_{source}} \xrightarrow{\overline{\mathrm{e\_s}}(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \text{asynch-send}, \mathrm{e\_s})}_{\mathrm{R_a}} \mathrm{x_{dest}}, \mathrm{y_{source}} \xrightarrow{\mathrm{e\_s}(?, \mathrm{L_b}, \mathrm{p}, \text{asynch-send}, \mathrm{e\_s})}_{\mathrm{R_{Mediun(La,Lb)}}} \mathrm{y_{dest}}$) ,$g_{i+1}>$, or

$<g_i$,($x_{source} \xrightarrow{\overline{e\_r}(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \text{asynch-receive}, \mathrm{e\_r})}_{R_{Mediun(La,Lb)}} x_{dest}, y_{source} \xrightarrow{e\_r(?, \mathrm{L_b}, \mathrm{p}, \text{asynch-receive}, \mathrm{e\_r})}_{Rb} y_{dest}$) ,$g_{i+1}>$

is a synchronized step of M and either

$(Ls_i, Lr_i, p_i, op_i, e_i)$ = $\mathrm{Ann_{synch}}$($x_{source} \xrightarrow{\bar{e}(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \text{synch-send}, \mathrm{e})}_{R_a} x_{dest}, y_{source} \xrightarrow{e(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \text{synch-send}, \mathrm{e})}_{R_b} y_{dest}$),

or $(Ls_i, Lr_i, p_i, op_i, e_i)$ =

$\mathrm{Ann_{asynch\text{-}send}}$($x_{source} \xrightarrow{\overline{e\_s}(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \text{asynch-send}, \mathrm{e\_s})}_{R_a} x_{dest}, y_{source} \xrightarrow{e\_s(?, \mathrm{L_b}, \mathrm{p}, \text{asynch-send}, \mathrm{e\_s})}_{R_{Mediun(La,Lb)}} y_{dest}$) , or

$(Ls_i, Lr_i, p_i, op_i, e_i)$ = $\mathrm{Ann_{asynch\text{-}receive}}$($x_{source} \xrightarrow{\overline{e\_r}\,(\mathrm{L_a}, \mathrm{L_b}, \mathrm{p}, \text{asynch-receive}, \mathrm{e\_r})}_{R_{Mediun(La,Lb)}} x_{dest}$,

$$(y_{\text{source}} \xrightarrow{\text{e\_r(?, }L_b\text{, p,asynch-receive,e\_r)}}_{\text{Rb}} y_{\text{dest}}) .$$

Example 3.3: A totally-ordered SR-sequence of model M in Fig. 3(a) is (L_2, L_3, *py*, *synchronous-synchronization*, *py_m*).(L_1, L_3, *px*, *synchronous-synchronization*, *px_m*). The sequence of synchronized steps of model M is:

$$(000, \quad 0 \xrightarrow{\overline{\textit{py\_m}}\textit{(L2,L3,py,synch-send,py\_m)}}_{R_2} 1, 0 \xrightarrow{\textit{py\_m(?,L3,py,synch-receive,py\_m)}}_{R_3} 1, \quad 011) ,$$

$$(011, \quad 0 \xrightarrow{\overline{\text{px\_m}}\text{ (L1,L3,px,synch-send,px\_m)}}_{R_1} 1, 1 \xrightarrow{\text{px\_m(?,L3,px,synch-receive,px\_m)}}_{R_3} 2, \quad 112) .$$

For model M in Fig. 3(c), a totally-ordered SR-sequence that includes the synchronization between L_1 and Medium(L_1, L_2) and the synchronization between Medium(L_1, L_2) and L_2 is (L_1, L_2, *px*, *asynch-send*, *px_m_s*).(L_1, L_2, *px*, *asynch-receive*, *px_m_r*). This sequence models a message sent asynchronously from L_1 to L_2 and the corresponding reception of this message by L_2. The sequence of synchronized steps of model M is:

$$(000, \quad 0 \xrightarrow{\overline{\textit{px\_m\_s}}\textit{ (L1,L2,px,asynch-send,px\_m\_s)}}_{R_2} 1, 0 \xrightarrow{\textit{px\_m\_s(?,L2,px,asynch-send,px\_m\_s}}_{R_{\text{Medium(L1,L2)}}} 1, \quad 101) ,$$

$$(101, \quad 1 \xrightarrow{\overline{\text{px\_m\_r}}\text{ (L1,L2,px,asynch-receive,px\_m\_r)}}_{R_{\text{Medium(L1,L2)}}} 2, 0 \xrightarrow{\text{px\_m\_r(?,L2,px,asynch-receive,px\_m\_r)}}_{R_3} 1, \quad 112) .$$

3.2 Partially-Ordered SR-sequences of the Model

Partially-ordered SR-sequences of the model have the same format as partially-ordered SR-sequences of the implementation – there is a local sequence for each ALTS component.

Definition 3.12: An event annotation for an event in a local sequence of an ALTS has the format *(sending ALTS, receiving ALTS, port, op, label, j, i)*, where the meanings of the fields are the same as those for events in implementation-based local sequences in Def. 2.6.

Definition 3.13: A *local* sequence of ALTS component L_i is a totally-ordered sequence of send and receive events (*(sending ALTS$_1$, receiving ALTS$_1$, port$_1$, op$_1$, label$_1$, j$_1$, 1), (sending ALTS$_2$, receiving ALTS$_2$, port$_2$, op$_2$, label$_2$, j$_2$, 2), ...*), executed by L_i, where *(sending ALTS$_k$, receiving ALTS$_k$, port$_k$, op$_k$, label$_k$, j$_k$, k)* denotes the k^{th}, k>0, event in the local sequence. A send (receive) event in a local sequence of ALTS L_i has L_i as the *sending* (*receiving*) *ALTS.*

A local sequence of ALTS component L_i is feasible for L_i if it corresponds to a sequence of transitions through L_i.

Definition 3.14: Let s_{Li} = ((sending $ALTS_1$, receiving $ALTS_1$, $port_1$, op_1, $label_1$, j_1, *1*), (*sending* $ALTS_2$, *receiving* $ALTS_2$, $port_2$, op_2, $label_2$, j_2, *2*), ... *),* be a local sequence of ALTS component L_i. Local sequence s_{Li} is *feasible for* L_i if L_i has a sequence of transitions $s_1 \xrightarrow{a_1}_{Ri} s_2 \xrightarrow{a_2}_{Ri} s_3 ..., s_i \in Q_i$, where state s_1 is the start state $q0_i$ of L_i, and for event (*sending* $ALTS_k$, *receiving* $ALTS_k$, p_k, op_k, $label_k$, j_k, *k*) and transition $s_k \xrightarrow{a_k}_{Ri} s_{k+1}$, *k*>0, of L_i, one of the following conditions is true:

- op_k is a send event and annotation a_k = (*sending* $ALTS_k$, *receiving* $ALTS_k$, p_k, op_k, $label_k$),
- op_k is a receive event and annotation a_k = (*?*, *receiving* $ALTS_k$, p_k, op_k, $label_k$);

otherwise, local sequence s_{Li} is *infeasible* for L_i.

Note that a receive event (*sending* $ALTS_k$, *receiving* $ALTS_k$, p_k, op_k, $label_k$, j_k, *k*) in local sequence s_{Li} specifies the IDs of both the receiving ALTS and the sending ALTS. Such an event is feasible if transition $\mathrm{s_k} \xrightarrow{\mathrm{a_k}}_{\mathrm{Ri}} \mathrm{s_{k+1}}$ of ALTS *Li* is a receive transition whose annotation a_k = (*?*, *receiving* $ALTS_k$, p_k, op_k, $label_k$) specifies the same receiving ALTS as the receive event, but only specifies '*?*' for the sending ALTS, i.e., the ID of the sending ALTS is ignored. Recall that the '*?*' in the transition annotation indicates that the receiver can be synchronized with any sending ALTS that can access port p_k.

Note also that only the behavior of component L_i is considered when determining the feasibility of a local sequence of L_i. For an event (*sending* $ALTS_k$, *receiving* $ALTS_k$, p_k, op_k, $label_k$, j_k, *k*) in a local sequence of L_i, the event index j_k of the thread that synchronizes with L_i is ignored. If this event is a receive event, then the ID of the sending ALTS is also ignored, as described above. This means that a feasible local sequence of L_i may not actually be allowed to occur when the constraints imposed on L_i by the other component ALTSs in M are considered. Recall that the transition annotations in an ALTS do not include event indices. A given transition in an ALTS can appear in many different positions, i.e., with many different event indices, in a local sequence of the ALTS.

The test generation algorithm defined later generates a set of local sequences for each of the ALTS components. If two generated local sequences of a component are equivalent, then only one of the sequences is used for testing. Two equivalent local sequences of an ALTS component *L* contain equivalent events. The fields of equivalent events of *L* have equal values, except possibly for the event indices of the threads that synchronize with *L*. This means that two equal events of *L* will both involve messages that are exchanged between *L* and some other ALTS *L'*, but one of the equal events of *L* may be synchronized with the i^{th} event of *L'* while the other is synchronized with the j^{th} event of *L'*, $i \neq j$. Note, however, that the labels of the two equal events of *L* capture the message values that are exchanged between *L* and *L'* and these labels must be the same. Hence, *L's* behavior after a sequence of equal events will be the same.

Definition 3.15: Let s_1 and s_2 be feasible local sequences of ALTS component *L*, where

$s_1 =$ ((sending ALTS_1^1, receiving ALTS_1^1, port_1^1, op_1^1, label_1^1, j_1^1, i_1^1), (sending ALTS_2^1, receiving ALTS_2^1, port_2^1, $\mathrm{op2}_2^1$, label_2^1, j_2^1, i_2^1),….), and

$s_2 =$ ((sending ALTS_1^2, receiving ALTS_1^2, port_1^2, op_1^2, label_1^2, j_1^2, i_1^2), (sending ALTS_2^2, receiving ALTS_2^2, port_2^2, $\mathrm{op2}_2^2$, label_2^2, j_2^2, i_2^2),….) .

Event $(sending\,ALTS_a^1, receiving\,ALTS_a^1, port_a^1, op_a^1, label_a^1, j_a^1, i_a^1)$, $a>0$, of s_1 is *equivalent to* event $(sending\,ALTS_b^2, receiving\,ALTS_b^2, port_b^2, op_b^1, label_b^2, j_b^2, i_b^2)$, $b>0$, of s_2 if $(sending\,ALTS_a^1 = sending\,ALTS_b^2)$ $\wedge (receiving\,ALTS_a^1 = receiving\,ALTS_b^2) \wedge (port_a^1 = port_b^2) \wedge (op_a^1 = op_b^2) \wedge (label_a^1 = label_b^2) \wedge (i_a^1 = i_b^2)$.

Equivalent local sequences of ALTS component *L* have equivalent events.

Definition 3.16: Let s_1 and s_2 be feasible local sequences of ALTS component *L*, where

$s_1 =$ ((sending ALTS_1^1, receiving ALTS_1^1, port_1^1, op_1^1, label_1^1, j_1^1, i_1^1), (sending ALTS_2^1, receiving ALTS_2^1, port_2^1, $\mathrm{op2}_2^1$, label_2^1, j_2^1, i_2^1), …) , and

$\mathrm{s}_2 =$ ((sending ALTS_1^2, receiving ALTS_1^2, port_1^2, op_1^2, label_1^2, j_1^2, i_1^2), (sending ALTS_2^2, receiving ALTS_2^2, port_2^2, $\mathrm{op2}_2^2$, label_2^2, j_2^2, i_2^2), …).

Sequence s_1 is *equivalent to* sequence s_2 if event $(sending\,ALTS_k^1, receiving\,ALTS_k^1, port_k^1, op_k^1, label_k^1, j_k^1, i_k^1)$ of s_1 is equivalent to event $(sending\,ALTS_k^2, receiving$ ALTS_k^2, port_k^2, op_k^2, label_k^2, j_k^2, i_k^2) of s_2, for all $k>0$.

Definition 3.17: A partially-ordered SR-sequence *Q* is defined as a tuple $(s_{L1}, s_{L2}, ..., s_{Ln})$, where s_{Li} is a local sequence of ALTS L_i, $1 \leq i \leq n$.

If a send event *s* in local sequence s_{Li} of ALTS $\mathrm{L_i}$ is synchronized with receive event *r* in local sequence s_{Lj} of ALTS L_j, then <*s*,*r*> is a synchronization pair. We assume that the synchronization pairs in a partially-ordered SR-sequence of the model satisfy properties SP1 – SP3 in Section 2.3.2. The algorithm in Section 6 for generating an SR-sequence of a model ensures that these properties are satisfied.

Example 3.4: Fig. 4(b) shows a partially-ordered SR-sequence of the synchronous message passing model in Fig. 4(c) and of the implementation in Fig. 4(a). For the implementation, the sending and receiving threads in the SR-sequence are *T1 – T4*; for the model, the sending and receiving ALTSs in the SR-sequence are *L1 – L4*.

Each synchronization pair in a partially-ordered SR-sequence forms a synchronization event in the format defined in Def. 3.1, which defines the format for the synchronization events in a totally-ordered SR-sequence. We will refer to these synchronization events as TO-events. If one of the send or receive events in a synchronization pair is executed by an ALTS modeling a medium, then the operation for the corresponding TO-event is *asynch-send* or *asynch-receive*; otherwise, the operation for the TO-event is a *synchronous-synchronization* between two non-medium ALTSs.

Definition 3.18: Partially-ordered SR-sequence $Q = (s_{L1}, s_{L2}, ..., s_{Ln})$ is feasible for model M if there exists a total ordering of the TO-events in *Q* that is consistent with the happened-before relation among the events in *Q* and that is feasible for M.

The happened-before relation in Definition 3.18 is the well-known happened-before causality relation formulated by [Lamport 1978]. Intuitively, an event e_1 *happened-before* another event e_2 in a sequence if e_1 could have affected e_2. If not e_1 happened-before e_2 and not e_2 happened-before e_1, then e_1 and e_2 are considered to be concurrent events. The feasible, totally-ordered sequences of M were defined in Def. 3.11. There may be more than one feasible total ordering of the TO-events that is consistent with the happened-before relation among the events in *Q*.

Example 3.5: For the partially-ordered SR-sequence in Fig. 4(b) there are 3 synchronization pairs <*s1*,*r1*>, <*s2*,*r2*>, and <*s3*,r3>. These pairs form the three TO-events (L_1, L_2, *p2*, *synchronous-synchronization*, *p2*), (L_2, L_3, *p3*, *synchronous-synchronization*, *p3*), and (L_4, L_3, *p3*, *synchronous-synchronization*, *p3*). The sequence of TO-events (L_1, L_2, *p2*, *synchronous-synchronization*, *p2*).(L_4, L_3, *p3*, *synchronous-synchronization*, *p3*).(L_2, L_3, *p3*, *synchronous-synchronization*, *p3*) is consistent with the happened-before relation among the events in the SR-sequence in Fig. 4(b) and is feasible for model

M in Fig. 4(c) based on Def. 3.11. The other consistent and feasible sequence of TO-events is (*L_4*, *L_3*, *p3*, *synchronous-synchronization*, *p3*).(*L_1*, *L_2*, *p2*, *synchronous-synchronization*, *p2*).(*L_2*, *L_3*, *p3*, *synchronous-synchronization*, *p3*).

As shown later, applying reachability testing to a model M requires us to draw a correspondence between events in two different feasible, partially-ordered SR-sequences of the model. The following definition shows that two events are equal if their annotations are equal.

Definition 3.19: Let *Q* and *Q′* be two feasible, partially-ordered SR-sequences of model M. Let *e* be an event in Q and *e′* an event in *Q′*. Events *e* and *e′* are equal, denoted as *e* = *e′*, if the annotations for *e* and *e′* are identical.

Intuitively, two feasible, partially-ordered SR-sequences of the model are equivalent if their events are equal.

Definition 3.20: Two feasible partially-ordered SR-sequences *Q* and *Q'* are equivalent, denoted as *Q* ≈ *Q'*, if there exists a one-to-one mapping *m* from the events in *Q* to the events in *Q'* that preserves event equality, i.e., $m(e_1) = e_2$ if and only if e_1 and e_2 are equal events.

Since equal events have equal annotations, and the event annotations for the events in an SR-sequence capture the event labels, the order of events, and the synchronization pairs in the SR-sequence, equivalent SR-sequences have events with the same labels in the same partial order, and have the same synchronization pairs.

3.3 Model-Based Testing with SR-sequences

In order to perform model-based testing with SR-sequences, it must be possible to compare the values of the messages passed between threads during an execution to the expected values specified as transition labels in an ALTS model. Message values for implementation messages are encoded by the *label* fields in the send and receive event annotations. The labels in an ALTS encode the expected message values. Different possible message values are represented by different labels, at some chosen level of abstraction. For example, in Figs. 3(a) and 3(b), we use the label "px_m" to represent sending or receiving message *m* on port *px*. The level of abstraction affects the size of the ALTS model and the types of errors that can be detected during model-based testing [Carver 1996].

The *Modern Multithreading* class library mentioned earlier automatically generates information about each send or receive event that is executed by the implementation, including the port name, the operation performed, and the IDs of the sending and receiving threads. However, the programmer must assist in generating labels for the exercised events. These labels are expected to match the labels that appear in the corresponding ALTS models. Event labeling in the *Modern Multithreading* library is performed using special event-labeler objects associated with ports.

Using programmer-generated event labels introduces the potential for "mapping errors" in the implementation. Event-labeler objects can be tested separately using normal testing techniques for objects. However, mapping errors that go undetected may allow an incorrect program to pass its tests, or cause a correct program to fail. The potential for errors in the generated labels is a limitation of our approach to mapping between the model and its implementation.

As a practical matter, it may be easier to work with totally-ordered SR-sequences than partially-ordered sequences. Totally-ordered SR-sequences are easier to represent, and it is easier to refer to properties like "the last event of a sequence" when the SR-sequence is totally-ordered. Thus, instead of dealing with some partially-ordered SR-sequence *s*, we may instead refer to a totally-ordered SR-sequence that is consistent with the causal ordering of events in *s*.

Lemma 3.1: Let *s* be a feasible, partially-ordered SR-sequence and *t* be a totally-ordered sequence that is consistent with *s*. Then *s* is feasible for CP iff *t* is feasible for CP [Tai and Carver 1996].

During model-based testing, we may need to determine whether a feasible SR-sequence of model M is feasible for implementation CP.

Definition 3.19: Let *s* be a feasible, totally- or partially-ordered SR-sequence of model M. Sequence *s* is feasible for implementation CP if an execution of CP with input *inputs(s)* can exercise sequence *s* and produce result *outputs(s)*.

The *Modern Multithreading* class library contains deterministic testing tools for determining whether a specified global SR-sequence *s* of M is feasible for a given implementation. Deterministic testing uses a "forced execution" of the implementation to determine whether *s* is feasible. The deterministic testing tool used to perform the case studies in this paper is described in [Carver and Tai, 2006; Carver and Lei 2010a, Carver and Lei 2010b]. The tool allows the global SR-sequence to be in either total- or partial-order format. The abstract input and output events in test sequence *s* must be translated into concrete inputs and expected outputs of the implementation before the test is executed.

4. Implementation Relations

Test generation begins with an abstract model M comprised of a set of ALTSs $\{L_1, L_2, \ldots, L_m\}$, and a concrete implementation CP of M with concurrent threads $\{P_1, P_2, \ldots, P_n\}$. We assume that a mapping exists between the ALTSs in M and the threads in CP, but we allow some flexibility in this mapping. In some cases, two or more ALTSs in M may be composed to create a single ALTS that is mapped to a thread in CP. In other cases, some ALTSs in M may not be mapped to any thread in CP. For example, M may contain an ALTS that models the behavior of a reliable communication medium for which there is no equivalent thread in CP. Likewise, M may contain an ALTS that models a component of the environment that supplies program inputs. To simplify our presentation, we assume that the number *n* of threads equals the number *m* of ALTSs and that thread P_i of CP is mapped to ALTS L_i of M. We also assume the alphabets of event labels for P_i and L_i are intended to be the same.

4.1 Implementation Relation M $\leq_F$ CP

The correctness of an implementation CP can be defined in terms of an implementation relation that is required to hold between CP and the ALTS model M of CP. The set of all feasible, finite, partially-ordered SR-sequences of model M is denoted by $Feasible_M$. The set of all feasible, finite, partially-ordered SR-sequences of implementation CP is denoted by $Feasible_{CP}$.

A global implementation relation that is often used for test generation is denoted by M $\leq_F$ CP.

Definition 4.1: M $\leq_F$ CP $\equiv_{\text{def}}$ $Feasible_M \subseteq Feasible_{CP}$.

Relation M $\leq_F$ CP requires each feasible sequence *s* of model M to be feasible for implementation CP. However, CP may have feasible sequences that are not feasible for M. This relation indicates that M is perhaps incomplete and thus is *extended* by CP, i.e., CP adds behavior that is not in M, but all of the behaviors of M are still allowed by CP [Brinksma 1988; Brinksma and Scollo 1986; Brinksma and Scollo 1987].

Example 4.1: For model M in Fig. 3(a), sequence (L_2, L_3, *py*, *synchronous-synchronization*, *py_m*).(L_1, L_3, *px*, *synchronous-synchronization*, *px_m*) is feasible. There is no feasible sequence of M in which LTSs L_1 and L_3 synchronize before LTSs L_2 and L_3. Implementation CP in Fig. 3(b) allows the synchronizations to occur in either order, but relation M $\leq_F$ CP still holds.

In Def. 4.1, sequence *s* is a sequence in $Feasible_M$, hence the feasible sequences of CP are the sequences that can be exercised by CP when CP is executed *with inputs modeled by M*. Assume M $\leq_F$ CP holds. If CP were to be executed with an input that is not modeled by M, then CP might exercise a sequence that is not feasible for M, but we only consider inputs modeled by M when we verify M $\leq_F$ CP. In the remainder of this paper, the term "feasible sequences of CP" refers to the sequences of CP that are feasible for inputs that are modeled by M. It may also be true that if the alphabets of M and CP were revised to, say, model program behavior at a lower level of abstraction, then errors in CP would be exposed, preventing relation M $\leq_F$ CP from holding. In general, an implementation relation is verified with regard to a given level of abstraction in the specification model. A "verified" implementation may still contain errors that are masked by the abstractions.

4.2 A Local Implementation Relation for $\leq_F$

The implementation relation in Def. 4.1 is for the full model M and its implementation CP. In this section, we define an implementation relation for an individual thread P_i in CP and the ALTS $L_i = <Q_i, E_i, A_i, R_i, q0_i, N_i>$ in M to which P_i is mapped. The definition of a feasible local sequence of ALTS L_i was given in Def. 3.14. The set of all feasible, finite, local SR-sequences of ALTS L_i is denoted by $Feasible_{Li}$.

As we mentioned in Section 3.2, a feasible local sequence of L_i may not actually be allowed to occur when the constraints imposed on L_i by L_i*'s* environment in M are considered. For example, L_i may allow two messages to be received in either order, while L_i*'s* environment may require the first message sent to L_i to be received before the second message sent to L_i can be sent.

Definition 4.2: A feasible local SR-sequence t_{Li} of ALTS L_i is *constrained with respect to model M* if there exists a feasible, partially-ordered SR-sequence $Q = (s_{L1}, s_{L2}, ..., s_{Ln})$ of M such that $t_{Li} = s_{Li}$. The set of constrained sequences of L_i with respect to model M is denoted *Constrained-Sequences*(L_i, M), or just *Constrained-Sequences*(L_i) when M is understood.

Constrained-Sequences(L_i) captures the constraints imposed on L_i by the other ALTSs in M.

Example 4.2: In Fig. 5, Constrained-Sequences(L_2) contains sequence (L_1, L_2, *a*, *receive*, *a*, *1*, *1*) and sequence (L_1, L_2, *a*, *receive*, *a*, *1*, *1*).(L_1, L_2, *b*, *receive*, *b*, *2*, *2*). Constrained-Sequences(L_2) does not contain a sequence that starts with a receive event for port *b*, even though L_2 itself is allowed to execute such a receive event first. This is because there is no feasible sequence of M that begins with a synchronization between L_2 and L_1 on port *b*.

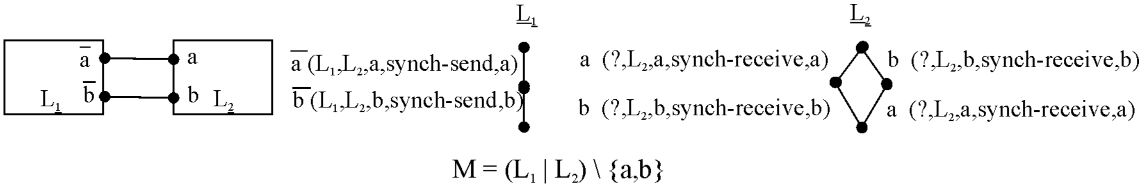

$\bar{a}$
$\bar{b}$
L_1
a
b
L_2
L_1
$\bar{a}$ (L_1,L_2,a,synch-send,a)
$\bar{b}$ (L_1,L_2,b,synch-send,b)
L_2
a (?,L_2,a,synch-receive,a)
b (?,L_2,b,synch-receive,b)
b (?,L_2,b,synch-receive,b)
a (?,L_2,a,synch-receive,a)
M = (L_1 | L_2) \ {a,b}

Figure 5. Local sequences of an ALTS.

The annotations on the events in a local test sequence s_{Li} specify the interactions that occur between L_i and its environment when the events in s_{Li} are exercised. If L_i exercises a receive (send) event then the environment should exercise a matching send (receive) event when the test sequence is executed. An environment of L_i that interacts as specified by the annotations in s_{Li} is referred to as a *conforming environment* of sequence s_{Li}.

Definition 4.3: A local sequence s_{Li} in Constrained-Sequences(L_i, M) is feasible for implementation thread P_i if P_i can exercise sequence s_{Li} when P_i is executed with a conforming environment of s_{Li}.

A procedure for checking the feasibility of a constrained, local test sequence for an implementation thread is given in Section 8. In this procedure, when determining the feasibility of local sequence s_{Li} for the thread P_i that implements L_i, synchronizations between P_i and the other threads in CP do not actually occur. Instead, a conforming test environment, in the form of a test driver, simulates P_i*'s* environment in CP by supplying the send and receive events that match the (annotated) events executed by P_i in local sequence s_{Li}.

Theorem 4.1: Let $Q = (s_{L1}, s_{L2}, ..., s_{Ln})$ be a feasible, partially-ordered SR-sequence of model M. Sequence Q is feasible for implementation CP iff for every $i \in \{1..n\}$ constrained local sequence s_{Li} is feasible for thread P_i.

Proof: See Section A.1 in the Appendix.

Based on Theorem 4.1, each thread in CP can be tested separately with the constrained local sequences of its corresponding ALTS component model, instead of testing all the threads together with all the feasible sequences of model M. Accordingly, a local implementation relation was defined in [Carver and Lei 2013] for ALTS-Thread pairs (L_i, P_i):

Definition 4.4: $L_i \leq_F P_i \equiv_{\text{def}}$ for any sequence s_{Li} in Constrained-Sequences(L_i): s_{Li} is feasible for P_i.

Local implementation relation $L_i \leq_F P_i$ is used in the following theorem, which is the basis for local testing:

Theorem 4.2: M $\leq_F$ CP iff for every $i \in \{1..n\}$ $L_i \leq_F P_i$.

Proof: See Section A.1 in the Appendix.

According to Theorem 4.2, the implementation relations between the individual threads in CP and the ALTSs in M can be verified separately in order to verify the implementation relation between M and CP. Testing each ALTS-Thread pair separately is more efficient in cases where the local sequences of an ALTS L_i are repeated many times, perhaps even an exponential number of times, in the feasible, global sequences of M.

We point out that our approach is local in the sense that it tests an individual thread P_i separately; however, as we will see, our approach derives the constrained local sequences for testing P_i from M not L_i. Thus, our approach is not local in the stronger sense that it tests an individual thread P_i with test sequences that are generated from ALTS L_i and only ALTS L_i. Note that L_i may have feasible local sequences that are not constrained. Using these local sequences to test P_i may cause spurious test failures — if these local sequences are not feasible for P_i, it does not imply that M $\leq_F$ CP is violated. Likewise, if these local sequences are infeasible for P_i, as expected, but they cause runtime assertions in P_i to fail, or exceptions to be thrown, during test execution, it does not imply that P_i has faults.

4.3 Implementation Relation M $\approx_F$ CP

In this section we define two more global implementation relations. The first relation reverses the roles that M and CP have in Definition 4.1 [Brinksma 1988; Brinksma and Scollo 1986; Brinksma and Scollo 1987; Chung et al. 2001]:

Definition 4.5: CP $\leq_F$ M $\equiv_{\text{def}}$ $\mathit{Feasible}_{CP} \subseteq \mathit{Feasible}_M$.

Relation CP $\leq_F$ M requires each feasible SR-sequence *s* of implementation CP to be feasible for model M. However, M may have feasible sequences that are not feasible for CP. Note that the global testing technique in [Lei and Carver 2006] can be used to enumerate all of the feasible, global SR-sequences of CP and show that these sequences are also feasible for M. Thus, this technique can be used to verify CP $\leq_F$ M. However, there may still be sequences that are feasible for M but not CP, so M $\leq_F$ CP may not hold.

The final implementation relation combines relations M $\leq_F$ CP and CP $\leq_F$ M [Tai 1985].

Definition 4.6: M $\approx_F$ CP $\equiv_{\text{def}}$ (M $\leq_F$ CP and CP $\leq_F$ M).

Relation M $\approx_F$ CP requires each feasible sequence *Q* of M to be feasible for implementation CP, and each feasible sequence *Q* of CP to be feasible for M. By implication, a sequence *Q* that is infeasible for M should also be infeasible for CP, and vice versa.

Example 4.3: Returning to Figs. 3(b) and 3(c), M $\approx_F$ CP is not satisfied since the sequence (L_1, L_3, *px*, *synchronous-synchronization*, *px_m*).(L_2, L_3, *py*, *synchronous-synchronization*, *py_m*) is feasible for CP, but not for M.

4.4 A Local Implementation Relation for $\approx_F$

A local version of relation $\approx_F$ can be defined by first defining a local relation for CP $\leq_F$ M. This relation mirrors the local relation for M $\leq_F$ CP in Section 4.2. We first define the set Constrained-Sequences(P_i), which captures the constraints imposed on thread P_i by the other threads in CP. We then define the feasibility of a local sequence in Constrained-Sequences(P_i) with respect to component ALTS L_i. These definitions appear as Definitions 4.7 and 4.8 in Section A.1 of the Appendix. The Appendix also contains Theorem 4.3, which shows that if sequence $Q = (s_{P1}, s_{P2}, ..., s_{Pm})$ is feasible for CP, then local sequence s_{Pi} is feasible for ALTS L_i if and only if sequence Q is feasible for M, for every $i \in \{1..n\}$.

A local version of CP $\leq_F$ M can now be defined on Thread-ALTS pairs (P_i, L_i).

Definition 4.9: $P_i \leq_F L_i \equiv_{def}$ for any sequence s_{Pi} in Constrained-Sequences(P_i): s_{Pi} is feasible for L_i.

Theorem 4.4: CP $\leq_F$ M iff for every $i \in \{1..n\}$, $P_i \leq_F L_i$.

Proof: See Section A.1 in the Appendix.

The local relations for $L_i \leq_F P_i$ and $P_i \leq_F L_i$ are used to define a local relation for $\approx_{F.}$

Definition 4.10: $L_i \approx_F P_i \equiv_{def}$ for every $i \in \{1..n\}$, $L_i \leq_F P_i$ and $P_i \leq_F L_i$.

The following Theorem shows that the local implementation relation $\approx_F$ between the individual threads in CP and the component ALTSs in M can be verified separately in order to verify the global relation M $\approx_F$ CP.

Theorem 4.5: M $\approx_F$ CP iff for every $i \in \{1..n\}$, $L_i \approx_F P_i$.

Proof: See Section A.1 in the Appendix.

5. Stateless Reachability Testing

In this section, we provide an overview of the stateless reachability-testing algorithm in [Lei and Carver 2006]. This algorithm heretofore has only been applied to concrete implementations. Reachability testing assumes that the implementation under test is closed, i.e., that the environment with which the program interacts is modeled by part of the implementation.

5.1. Message Races

Assume receive event *r* is synchronized with send event *s* during some program execution. Informally, there exists a *message race* or simply a *race* between send event *s* and another send event *s'* with regard to a receive event *r* if *r* could have received the message sent by *s'* instead of the message sent by *s*. For example, in the sequence in Fig. 4(b), there exists a message race between send events s_2

and s_3 with regard to r_3, since r_3 could have received the message sent by s_2 instead of the message sent by s_3. The send events that have a race with the send event *s* that is synchronized with *r* comprise the *race set* of *r*.

In order to identify the races in an execution and compute the race set of a receive event, an *OpenList* is computed for each receive event.

Definition 5.1: Let *Q* be the *SR*-sequence exercised by an execution of a concurrent program *CP* with input X, where CP uses *asynchronous message-passing*. The *OpenList* of a receive event *r* in *Q* contains a single port, which is the source port of *r*.

The *OpenList* of a receive event in a program that uses synchronous message-passing depends on whether or not the receive event is a receive-alternative of a selective wait.

Definition 5.2: A receive-alternative is said to be *open*, and thus selectable, if it does not have a guard condition, or if the value of the guard condition is true. Otherwise, the alternative is said to be *closed* and it cannot be selected.

We assume that all of the alternatives of a selective wait are receive alternatives, and that no two receive-alternatives access the same port.

Definition 5.3: Let *Q* be the SR-sequence exercised by an execution of a concurrent program CP with input X, where CP uses *synchronous message-passing*. The *OpenList* of a receive event *r* that is a receive-alternative of a selective wait is the list of ports that had open receive-alternatives when *r* was selected. The *OpenList* of a receive event *r* that does not occur inside a selective wait contains only the source port of *r*.

Definition 5.4: Let *Q* be the SR-sequence exercised by an execution of a concurrent program CP with input X. A send event *s* in Q is said to be *open* at *r* if *s.port* is in the *OpenList* of *r*.

In order to accurately determine all the races in an execution, the program's logic must be analyzed. Detecting races using static analysis is undecidable for arbitrary programs [Bernstein 1966] and is NP-complete for even very restricted classes of programs, e.g., those containing no branches [Taylor 1983].

However, for the purpose of reachability testing, we only need to consider a special type of race, called a lead race. Lead races can be identified by analyzing the SR-sequence of an execution, i.e., without analyzing the program's logic.

Definition 5.5: Let *Q* be the SR-sequence exercised by an execution of a concurrent program CP with input X. Let *s* be a send event and *r* be a receive event in *Q* such that <*s, r*> is a synchronization pair. Let *s'* be another send event in *Q*. There exists a lead race between send events *s'* and *s* with respect to *r* in *Q* if *s'* and *r* can be synchronized with each other during an alternative execution of CP with input X in which all the events that happen-before *s'* or *r* in *Q*, as well as the synchronizations between these events, are replayed.

Def. 5.5 requires all the events that can potentially affect *s'* or *r* in *Q* to be replayed, i.e., forced to execute in the same order that they were executed in *Q*. This guarantees that *s'* and *r* can be exercised in the alternative execution. In the rest of this paper, a race is assumed to be a lead race unless otherwise specified. During reachability testing, a vector timestamp is assigned to each send and receive event. The timestamps can be used to determine the *happened-before* relation between events [Fidge 1988; Mattern 1988].

Definition 5.6: Let *Q* be an SR-sequence. Let *s* be a send event and *r* be a receive event in *Q* such that <*s, r*> is a synchronization pair. The *race set* of *r* in *Q*, denoted as *race_set(r, Q)* or *race_set(r)* if *Q* is implied, is the set of send events in *Q* that have a race with send event *s* with regard to *r*. Formally, *race_set(r, Q) = {s' ∈ Q | there exists a race between send events s' and s with respect to r}*.

Proposition 1 describes how to compute the race set of a receive event.

Proposition 1: Let *Q* be an SR-sequence. A send event *s'* is in the race set of a receive event *r* in *Q* if (1) *s'* is open at *r*; (2) not *r* happened-before *s'*; (3) if <*s', r'*> is a synchronization pair, then *r* happened-before *r'*; and (4) if a send event *s''* has the same source and destination thread as *s'* but happened-before *s'*, then there exists a receive event *r''* such that <*s'', r''*> is a synchronization pair and *r'* happened-before *r''*.

Conditions (2) and (3) in Proposition 1 ensure that changing the sender for receive event *r* cannot affect whether send *s'* is made or receive *r* occurs, respectively. Condition (4) reflects the assumed *FIFO* message ordering scheme for asynchronous and synchronous message-passing. That is, since messages sent from the *same thread* to the same port are received in the order that they are sent, the message sent by *s'* cannot be received by *r* unless the message *s''* sent earlier was received before *r*.

Details about how to compute race sets can be found in [Tai 1997; Lei and Carver 2006]. Basically, each of the send events in a given sequence *s* is visited one-by-one. For each send event *snd*, if *snd* is received by $Thread_i$, then a search is performed for receive events in sequence *s* that were executed by $Thread_i$ and that could receive send event *snd* based on the ports of the receive events and the happened-before relation between the events. The complexity of the algorithm for computing race sets is O(D*R), where D and R are the total numbers of send and receive events, respectively, in *s*. The algorithm is also O(D+F), where F is the sum of the sizes of the race sets associated with receive events in *s*, which is bounded by O(D*R).

5.2 Race Variants

A *race variant* of an SR-sequence *Q* is a (partially-ordered) prefix of *Q* in which the outcome of one or more races has been changed. The variant represents an SR-sequence that could have been executed by CP.

The formal definition of race variant in Definition 5.7 below references the "control-structure" of an event. Informally, the control-structure of event *e* contains all the events, as well as the synchronizations between the events, that could possibly *control* whether or not event *e* is executed. Fig. 6 shows a sequence *Q* for an asynchronous message passing execution. Send event s_3 happened-before receive event r_3 since (s_3, r_3) is a synchronization pair. Receive event r_1 also happened-before r_3 since T2 executes r_1 before it executes r_3. However, there is a subtle distinction between the happened-before relation for s_3 and r_3 and that for r_1 and r_3. Let *R* be the receive statement that was executed when r_3 occurred. Then, whether or not statement *R* is executed, i.e., whether or not r_3 occurs, may depend on the control exerted by the execution of events s_1 and r_1, but it does not depend on the execution of s_3. Therefore, events r_1 and s_1 are considered to be in the "control-structure" of r_3, whereas s_3 is not in the control-structure of r_3.

Definition 5.7: Let *e* be an event exercised by a thread *T* in a sequence *Q*. Then, the control-structure of *e* in *Q*, denoted as *c-struct(e, Q)*, is empty if *e* is the first event exercised by *T*; otherwise, the control structure contains the event *f* that *T* exercised immediately before *e*, and all the events that happen-before *f*, including the synchronizations between these events.

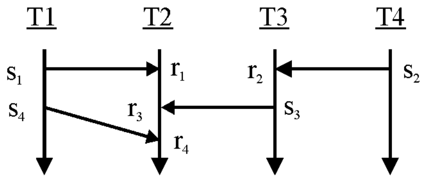


Figure 6. SR-sequence Q.

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

Definition 5.8 [Lei and Carver 2006]: Let *Q* be a sequence. A race variant *V* of *Q* is an SR-sequence that satisfies the following conditions:

(1) There exists at least one receive event *r* in both *Q* and *V* such that *send(r, Q)* ≠ *send(r, V)*

(2) Let *r* be a receive event in *Q* and *V*. If *send(r, Q)* ≠ *send(r, V)*, then *send(r, V)* must be in *race_set(r, Q)*

(3) Let *e* be a send or receive event in *Q*. Then, *e* is not in *V* if and only if there exists a receive event *r* in *Q* such that *r* ∈ *c-struct(e)* and *send(r, Q)* ≠ *send(r, V)*.

Note that condition (3) ensures that race variant *V* is always feasible, i.e., *V* can be exercised by at least one program execution, regardless of the program's control and data flow. This is because after the sending partner of a receive event *r* is changed, the third condition requires all the events whose existence might be affected by this change to be removed from *V*. This is a conservative approach since some of the events that are removed may not actually be affected.

5.3 Reachability-Testing Algorithm

In this section, we describe and illustrate the stateless reachability-testing algorithm from [Lei and Carver 2006]. This algorithm can be applied to programs that use send and receive statements, and also to selective wait statements that allow a thread to wait for a message to be sent to any of two or more ports.

5.3.1 Assumptions about the Implementation

The reachability-testing algorithm requires that two or more executions of a concurrent program with the same input and the same SR-sequence always produce the same result. The result of an execution includes the output and termination condition of the execution. The possible types of abnormal termination include deadlock, exceptions, and assertion violations, among others. When this requirement is satisfied, an SR-sequence provides sufficient information for replaying all or part of an execution, which is required during reachability testing.

This requirement would not be satisfied by programs that, e.g., have uninitialized variables, or that access memory after it has been de-allocated. These types of memory errors can cause different results to be generated for two executions that have the same inputs and the same SR-sequences. If a memory error occurs, the error may or may not be detected during execution, and the error may or may not cause the execution to fail. In general, the potential for undetected execution errors limits the effectiveness of reachability testing and of our approach to verifying an implementation relation between a model and its implementation. Various tools and techniques have been developed for detecting memory errors and these tools can be used during testing to ensure that pass/fail verdicts are accurately assigned to test cases.

The reachability-testing algorithm explores all of the non-deterministic execution behaviors of a program by considering all of the non-deterministic message races in the SR-sequences of the program's executions. Threads that execute select statements of the following types can exhibit non-deterministic behavior that is not captured by analyzing the message races in an SR-sequence: select statements that make a selection between two send statements, or between a send statement and a receive statement; and select statements that make a selection between two receive statements that access the same port. However, these types of selections are typically not allowed by implementation-level, message-passing constructs, so they are not likely to be encountered in practice. The reachability-testing algorithm assumes that selections are made between receive statements only, and that the receive statements in a select statement all access different ports.

For a concurrent program CP that uses send and receive statements and selective wait statements, and that satisfies the above assumptions about memory errors and select statements, [Tai et al. 1991] proved that the result of an execution of CP with input X is determined by CP, X, and the SR-sequence of the execution.

5.3.2 Algorithm

Fig. 7 shows the general reachability-testing algorithm from [Lei and Carver 2006]. This algorithm, when applied to a concurrent program CP, exercises every feasible, partially-ordered SR-sequence of CP with input X exactly once.

```
ALGORITHM Reachability-Testing (CP: a concurrent program; X: an input of CP) {
1.   let variants be an empty set;
2.   collect an SR-sequence $Q_0$ by executing CP with input X non-deterministically;
3.   let $V_0$ be the special empty variant that contains no events
4.   variants = GenerateVariants($Q_0$, $V_0$)
5.   while (variants is not empty) {
6.      withdraw a variant V from variants;
7.      collect SR-sequence Q by conducting a prefix-based test run with V;
8.      variants = variants $\cup$ GenerateVariants(Q, V);
9.   }
10.}
```

Figure 7. Reachability testing algorithm.

We illustrate this algorithm by applying it to program CP in Fig. 8(a). In program CP, thread *T2* receives messages from threads *T1* and *T3* and sends them each a reply. Fig. 8(b) shows a scenario in which reachability testing is applied to CP.

```
T1                     T2                       T3
p2.send(1);            m1 = p2.receive();       p2.send(2);
s = p1.receive();      m2 = p2.receive();       s = p3.receive();
                       p1.send( m2-m1);
                       p3.send( m2-m1);
```

(a)

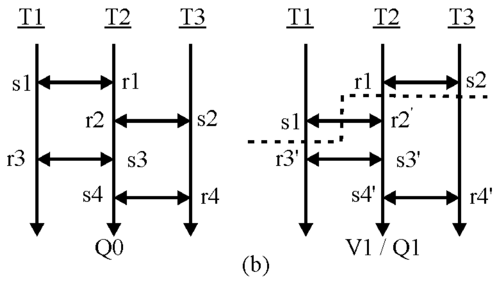


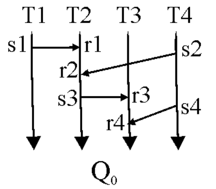
Figure 8. A reachability testing scenario.

Reachability testing begins by executing CP non-deterministically, which we assume exercises SR-sequence Q_0 (line 2 in Fig. 7). Procedure *GenerateVariants* (described below) generates the set of race variants of a collected sequence Q, denoted *Variants(Q)*. Each race variant is a partially-ordered SR-sequence.

In the first call to procedure *GenerateVariants*, the race in Q_0 between *s1* and *s2* is identified and the outcome of this race is changed to derive race variant V_1 of Q_0 (line 4 in Fig. 7). That is, V_1 is derived by changing the send partner of *r1* from *s1* to *s2* and removing event *r2*. Event *r2* must be removed since *r2* might not be executed after *r1* receives the message from *s2* instead of *s1*. This emphasizes the point that the variants of SR-sequence Q_0 are identified by analyzing Q_0, not the source code of program *CP*; hence, it is not possible to know what *CP* will do after *r1* receives a message from *s2* instead of *s1*. Variant V_1 is used to conduct a prefix-based test run, which forces the events and synchronizations in the variant to be replayed and then allows the test run to proceed non-deterministically, i.e., without controlling which SR-sequence is exercised (line 7 in Fig. 7). Prefix-based testing with V_1 exercises sequence Q_1. In Fig. 8(b), the sequence Q_1 and the variant V_1 used to exercise it are shown in the same space-time diagram. The events in the variant are the events above the dashed line. No new variants will be derived by *GenerateVariants* from Q_1 (line 8), so the reachability-testing process stops. Note that Q_0 and Q_1 are all of the (partially-ordered) SR-sequences the example program can possibly exercise.

5.3.3 Computing Variants

The algorithm used by *GenerateVariants* to compute race variants can be described as follows. This algorithm builds a "race table" for a given SR-sequence *Q*. Each row of the race table represents a unique, partially-ordered, SR-sequence, which is race variant of *Q*. As an example, Fig. 9 shows an SR-sequence Q_0 and the race table for Q_0.



	r1	r3
V1	0	1
V2	1	0
V3	1	1

Race table for Q0

Figure 9. SR-sequence Q_0 and its race table.

The race table contains columns for receive events r_1 and r_3, which are the receive events in Q_0 whose race sets are non-empty. Three variants can be generated for Q_0. These variants V_1, V_2, and V_3 are represented by rows 1, 2, and 3, respectively, of the race table. The column values indicate how the send partners of the receive events are changed to create a variant. For example, for variant V_1 represented by row 1, the value 0 indicates that the send partner of r_1 is left unchanged. The value 1 indicates that the send partner of r_3 will be changed; the new send partner of r_3 will be s_4, which is the first (and only) send event in the race set of r_3. For the variant V_3 represented by row 3, the send partners of r_1 and r_3 are changed to s_2 and s_4, respectively.

Generating all of the rows of the race table ensures that no variants are missed; however, additional work must be done to ensure that no duplicate sequences are generated. For example, a single "color" bit is stored with each receive event in a variant or sequence. The color of a receive event is initially white. The color of an event is changed to black if the event is changed to create a variant, or the event happened-before a changed event. The color of a black event is never changed to white. Races for black events are ignored by excluding black events from the race table. The reason black events are excluded is that changing the send partner of a black event could cause the event to revert to its original send partner, causing a sequence to be duplicated.

Due to space limitations, we refer the reader to [Lei and Carver 2006] for details about procedure *GenerateVariants*. We stress that every row in a race table represents a unique, partially-ordered race variant, and no analysis of the source code is required to build a race table. Let R be the set of receive events in sequence Q whose race sets are non-empty. The time complexity of the algorithm for variant generation is $O(|R|^2 * |V|)$, where V is the set of race variants for Q.

5.3.4 Complexity and Correctness

The time complexity of reachability testing is $O(n * |E_{max}|^2 * |V_{max}|)$, where n is the number of possible SR-sequences, $|E_{max}|$ is the maximum number of events in an SR-sequence, and $|V_{max}|$ is the maximum number of variants for an SR-sequence. Note that the empirical studies reported in Section 9 show that reachability testing for ALTSs is 7 – 10 times faster than for real programs, since analyzing LTS models has much less overhead than performing controlled executions of real programs. The space required for storing an SR-sequence collected during an execution, or for storing a variant of an SR-sequence, is linear in the length of the sequence. Storage is required for variants that have been generated but not executed; however, the number of these variants is typically much smaller than the total number of variants. In addition, variants can optionally be stored on disk and accessed using disk caching techniques that have low overhead. In the case of the DME program mentioned in Section 1, reachability testing exercised 1.45 billion sequences, but the maximum number of variants that needed to be stored at any one time was only 1,443.

Theorem 5.1**:** Given a closed concurrent program CP and an input X, if every execution of CP with input X terminates, then algorithm *Reachability-Testing* exercises every partially-ordered SR-sequence of CP with input X exactly once.

Proof: [Lei and Carver 2006].

If there is an execution of CP that does not terminate because of a cycle in CP's state space, then CP has infinite length SR-sequences, which makes it impossible to exercise every SR-sequence of *CP*. Algorithm *Reachability-Testing* can still be applied as long as some mechanism is used to ensure that every execution of CP terminates. Further discussion about state space cycles and termination appears in Section 6.2.

Reachability testing is implemented in a tool call *RichTest* [Lei and Carver 2006], which is part of the *Modern Multithreading* class library [Carver and Tai 2006; Carver and Lei 2010b]. A number of other verification tools have been developed that, like our *RichTest* reachability-testing tool, systematically explore the state space of a C or Java implementation by controlling its execution. These tools use partial-order reduction to avoid exercising redundant interleavings of the same partial ordering of events [Godefroid 1997]. Partial-order reduction exploits the commutativity of independent transitions and conducts a selective search in which only a subset, called a persistent set, of the enabled transitions in a global state are explored. A comparison between reachability testing and partial order reduction can be found in [Carver and Tai 2006].

6. Using Reachability Testing to Verify M $\leq_F$ CP with Global Test Sequences

In this section, we describe how the stateless reachability-testing algorithm *Reachability-Testing* described in Section 5 can be used to derive a global test oracle for an ALTS model.

6.1 Message Races

In order to identify the races and compute the race sets of the receive events in an SR-sequence of model M, an *OpenList* is computed for each receive event in an SR-sequence.

Definition 6.1: Let *Q* be an SR-sequence of M and let *r* be a receive event that is executed in state *s* of ALTS component *L* of M. The *OpenList* of *r* is a list of annotations, one for each of the transitions of state *s* in *L*.

The *OpenList* of *r* captures the events that ALTS *L* was willing to execute when *r* was executed. These are simply the transitions of state *s*.

Definition 6.2: Let *Q* be an SR-sequence of M. A send event *snd* in Q is said to be *open* at *r* if *snd.annotation* and the annotation for any receive event in the *OpenList* of *r* satisfy relation *Compatible* in Defs. 3.3 - 3.5 of Section 3.1.

The definitions of race sets and race variants for an SR-sequence of model M are the same as those defined in Sections 5.1 and 5.2 for an SR-sequence of a concrete implementation.

6.2 Restrictions

In order to apply algorithm *Reachability-Testing* to a model M of implementation CP, model M must be closed, i.e., CP's environment must be modeled as part of M. Techniques for automatically or semi-automatically closing an open model or implementation are described in [Colby et al. 1998; Hone at al. 2002; Hughes and Bultan 2008; Ioustinova et al. 2002; Parizek and Plasil 2007; Tkachuk and Dwyer 2003].

If M has a cycle in its state space, then *Reachability-Testing* will not terminate when it is applied to M. A similar problem is encountered by other test generation techniques that try to generate all the possible test sequences of cyclical models. To avoid infinite test suites, some method must be used to select a subset of M's sequences. The resulting test sequences, while sound, cannot be used to verify implementation relation M $\leq_F$ CP.

Test selection may be based on, e.g., guidance from the person doing the testing [Information Technology 1991; Feijs et al 2002; Tretmans and Brinksma 2003], or coverage criteria [Ammann and Offutt 2008]. One common approach is to limit the depth or duration of the state search. In our experience, it is not easy to bound the depth or duration of a state search that encounters many cyclic paths. Another approach is to modify cyclic model M in order to create an acyclic model M'. Model builders can apply their domain knowledge to control precisely how many iterations of the state space cycles are to be performed. Test sequences generated from acyclic model M' can be used to verify implementation relations between M' and a corresponding modified version of implementation CP. In the test generation procedures defined below, we assume that model M is acyclic, or has been made acyclic, so that the implementation relations defined in Section 4 are verifiable.

As we mentioned in Section 5.3.1, the reachability-testing algorithm in [Lei and Carver 2006] restricts the types of non-deterministic selections that a thread or component ALTS is permitted to make. Let *s* be a state in an ALTS component. State *s* must satisfy the following restrictions:

(R1) If *t* is a send-transition of state *s*, then *t* is the only transition of state *s*.

(R2) For any two receive-transitions t_1 and t_2 of state *s*, *t1.annotation* ≠ *t2.annotation*.

A non-deterministic selection between two send transitions, or between a send and receive transition is typically not supported by implementation-level programming constructs. A non-deterministic selection in an ALTS component model between two receive transitions that have the same annotation corresponds to a *select* statement in an implementation thread in which there are two receive alternatives that access the *same port*: select p.receive(); or p.receive(); end select. The executed receive events will have the same event annotations. We do not find these types of selections to be useful at the implementation level and, as we mentioned in Section 5.3.1, they are not allowed by the reachability-testing algorithm.

Although these types of non-deterministic selections are not permitted in the component models used for test generation, they may be useful earlier in the modeling process, e.g., for modeling design decisions that are to be made at some later point in development. Making a design decision amounts to a *reduction* of the non-determinism in the specification model [Brinksma 1988; Chung et al. 2001]. We do not discourage this type of design-level non-determinism. It can be used in earlier stages of modeling as long as this non-determinism is reduced before the model is used to generate test sequences for the implementation threads. This reflects the fact that we expect all of the design decisions for the threads to be made before the threads are tested.

When restrictions (R1) and (R2) are satisfied, the only source of non-determinism in an ALTS model is the order in which racing messages are received, and this order is determined by the model and the SR-sequence.

Theorem 6.1: Let *Q* be a feasible, partially-ordered SR-sequence of a model M that is comprised of a set of component ALTSs {L_1, L_2, …, L_n}. Assume that every state in each component ALTS satisfies restrictions (R1) and (R2). Then each selection that a component ALTS makes is specified by *Q* and is deterministic.

Proof: See Section A.2 in the Appendix.

6.3 Generating Global Test Sequences for M $\leq_F$ CP

In this section, we describe the changes that are required in order to apply stateless reachability testing to an ALTS model instead of an implementation. The major change is that an SR-sequence is generated by composing LTSs instead of by executing threads. Steps 2 and 7 of algorithm *Reachability-*

Testing (Fig. 7) collect an SR-sequence of M. The SR-sequence collected is generated one synchronization step at a time by composing the ALTs of M according to Def. 3.10 of Section 3.1.

Fig. 10 shows algorithm *GenerateSequence* for generating an SR-sequence of M. The algorithm inputs model M and the global starting state *startState* in M of the SR-sequence to be generated, and outputs a partially-ordered SR-sequence Q of M. *GenerateSequence* begins by finding a compatible pair <*s,r*> of send and receive transitions that are enabled in the *startState* of the generated sequence. In step 2 of algorithm *Reachability-Testing*, the value used for *startState* will be the global start state of M. The annotations for send event *s* and its compatible receive event *r* are added to the partially-ordered-ordered SR-sequence *PO*, with event indices that reflect the positions of *s* and *r* in their respective local sequences. The destination state for the synchronized step represented by <*s,r*> becomes the new *currentState*. *GenerateSequence* stops when *currentState* is a termination state of M. Since the SR-sequence format that is used for ALTS models is the same partial-order format used to represent SR-sequences of the implementation, no changes are required to procedure *GenerateVariants*.

```
ALGORITHM GenerateSequence (M: an ALTS model;
startState: starting state of generated sequence) {
    let global state currentState = startState;
    let PO be an empty, partially-ordered SR-sequence;
    while (currentState is not a terminal state) {
      synchPair <s,r> = CompatibleTransitions(currentState);
      PO.add(s.annotation, r.annotation);
      currentState = SynchronizedStep(currentState, <s,r>);
    }
    Output PO;
}
```

Figure 10. Algorithm to generate a partially-ordered SR-sequence of M.

For a variant *V* generated by *GenerateVariants*, no prefix-based test run is required to exercise *V* and collect a new SR-sequence of M (line 7 in Fig. 7). Instead, we first identify the local state that each component ALTS is in after the last event in *V*. The local source and destination states of the two components involved in each synchronization step of *V* can be stored and used to identify the local state of each component at the end of *V*. The local states are used to form the global *startState* of M that is passed as an argument to *GenerateSequence*. Algorithm *GenerateSequence* generates an SR-sequence *PO* of M that starts in *startState*. A new SR-sequence of M is collected by appending *PO* to *V*.

7. **Using Reachability Testing to Verify M $\approx_F$ CP with Global Test Sequences**

Test sequences generated to verify M $\leq_F$ CP can also be used to verify M $\approx_F$ CP, but some additional information is required about each receive event in the test sequences. Section 7.1 gives an overview of the required information and how it is used. Section 7.2 presents a test procedure that verifies M $\approx_F$ CP by verifying M $\leq_F$ CP and CP $\leq_F$ M.

7.1 Overview

Recall from Sections 5.1 and 6.1 that two send events *s1* and *s2* have a *race* with regard to a receive event *r* if either of the messages sent by events *s1* and *s2* can be received by event *r*. The racing send events of receive event *r* are used to compute the *race set* of *r*. Refer again to model M in Fig. 3(a) in Section 2.2. Model *M* has a single feasible SR-sequence *s*:

SR-sequence *s* of M:
(L_2, L_3, *py*, *synchronous-synchronization*, *py_m*), race set = {};
(L_1, L_3, *px*, *synchronous-synchronization*, *px_m*), race set = {}.

The race sets of the receive events in sequence *s* of M tell us the alternative behaviors that M permits implementation CP to execute when CP is presented with the inputs and send events in sequence *s*. In this case, the race sets for both receive events in sequence *s* are empty, indicating that neither receive event is permitted to be matched with a different send event.

Testing CP with sequence *s* will show that sequence *s* is a feasible sequence of implementation CP in Fig. 3(b). The corresponding SR-sequence executed by CP during testing is:

SR-sequence *s'* of CP:
(P_2, P_3, *py*, *synchronous-synchronization*, *py_m*), race set = {send event from P_1 to P_3};
(P_1, P_3, *px*, *synchronous-synchronization*, *px_m*), race set = {}.

The race set of the first receive event in sequence *s'* executed by CP is not the same as the race set of the first receive event in *s*. This is because the first receive event in *s'* is executed by a selective wait statement in P_3 that allows P_3 to receive its first message from thread P_1 instead of P_2. The first receive event in sequence *s* is executed by ALTS L_3, but L_3 does not allow a message from L_1 to be received first. This difference between the race sets for the first receive events in sequences *s* and *s'* captures the fact that there is a feasible sequence of CP in which thread P_3 starts by receiving a message from P_1, but this sequence is not in Feasible$_M$. Based on this difference in the race sets, we conclude that when implementation CP is executed with inputs modeled by M, CP can exercise a sequence that is not allowed by M, which means that the implementation relation M $\approx_F$ CP does not hold.

7.2 A Test Procedure for Verifying M $\approx_F$ CP with Global Test Sequences

A test procedure for verifying global implementation relation M $\approx_F$ CP is shown in Fig. 11. Model M is assumed to be acyclic so that relation M $\leq_F$ CP is verifiable.

```
Procedure Test$_{\approx F}$:
(a) Generate Feasible$_M$.
(b) For each test sequence s in Feasible$_M$:
    (b1) Perform deterministic testing on CP with sequence s and inputs(s)
         and assign a test verdict, which is either pass or fail. If CP fails with
         s, then testing halts.
    (b2) Let s' be the sequence executed by CP that corresponds to s. For each
         receive event r in s:
            Let r' be the corresponding receive event in s'. If (r.race_set = r'.race_set)
            then assign the verdict pass; else assign fail. If CP fails with s, then a
            failure has been detected and testing halts.
```

Figure 11. A global test procedure for M $\approx_F$ CP.

The first two steps of procedure Test$_{\approx F}$ are used for checking whether relation M $\leq_F$ CP holds. Step (a) is to generate the set Feasible$_M$ of feasible, *partially-ordered* sequences of M, using the stateless reachability-testing procedure described in Section 6.3.

In step (b1), deterministic testing is performed on CP with test sequence *s* of Feasible$_M$. The transition annotations are used to translate abstract test sequence *s* into a concrete, implementation-based SR-sequence. The deterministic testing tool described in Section 3.3 is used to determine whether SR-sequence *s* is feasible for implementation CP. The test verdict for deterministic testing with global test sequence *s* is assigned in step (b1) as follows: if sequence *s* is infeasible for CP or CP produces unexpected output(s) or terminates abnormally then the test *fails* else the test *passes*. If all the sequences in Feasible$_M$ pass then relation M $\leq_F$ CP holds.

Example 7.1: For model M in Fig. 3(a), Feasible$_M$ contains the sequence (L_2, L_3, *py*, *synch-receive*, *py_m*).(L_1, L_3, *px*, *synch-receive*, *px_m*). When this sequence is used to test implementation CP in Fig. 3(b), a verdict of *pass* will be assigned since this sequence is feasible for CP.

Assume that the last event of sequence *s* visits a termination state of M. After CP executes the last event in test sequence *s*, CP may try to execute an additional send or receive event instead of terminating. If so then CP can execute a sequence that is not feasible for M and the relation CP $\leq_F$ M does not hold. To specify the termination behavior of CP, we add event *stop* to the set *A* of annotations and use *stop* to denote normal termination. For a totally-ordered sequence $s = e_1.e_2....e_n$, we append *stop* to the end of sequence *s*. When $s = e_1.e_2....e_n.stop$ is used to test thread CP, we check whether CP terminates normally after executing event e_n. If CP does not terminate normally, then the test *fails*.

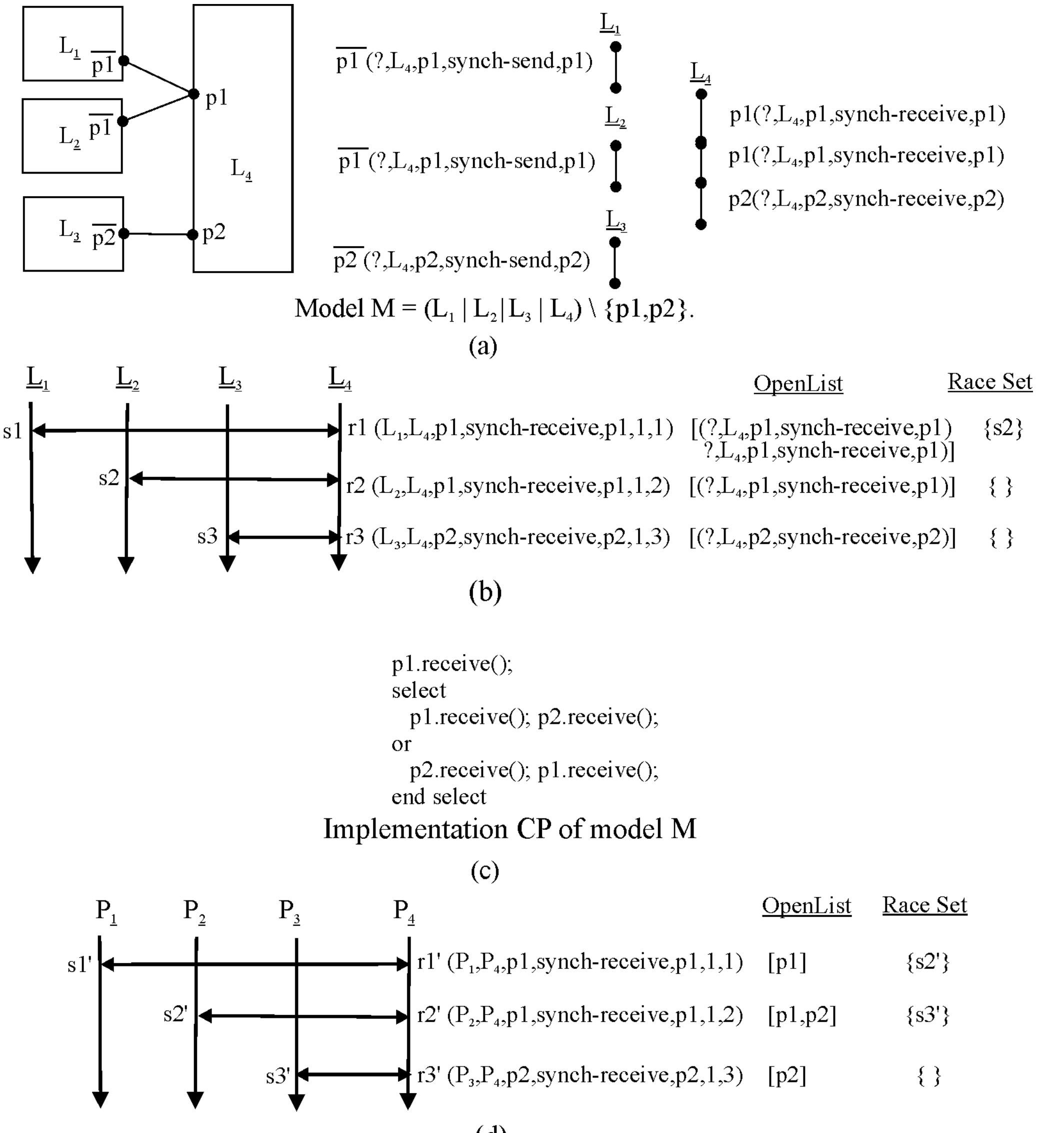
To check CP's termination behavior during deterministic testing, CP's behavior is monitored after the last event in the test. If CP attempts to execute an extra send or receive event, or CP terminates abnormally, then the test *fails*. If CP does not try to execute any extra events, but CP does not terminate within some user-specified time-out period, it is assumed that CP will not terminate, and the test *fails*. Otherwise, CP terminates normally and the test *passes*.

Step (b2) of procedure $\text{Test}_{\approx F}$ is used to check whether relation CP $\leq_F$ M holds. This step checks whether CP can exercise sequences that are not in Feasible_M, when CP is executed with inputs modeled by M and CP terminates normally as expected. Assume testing determines that test sequence *s* is feasible for CP and that CP exercises the corresponding sequence s' and displays the expected termination behavior. Let *r* be a receive event in test sequence *s*, and *r'* be the corresponding receive event in *s'* executed by CP. It is possible that when a thread in CP executed receive event *r'*, this thread was also willing to execute some other receive event r''. Step (b2) compares the race sets of receive events *r* and *r'* to determine whether CP can exercise any receive events besides *r'*, and whether these alternative receive events are allowed by M.

Algorithm *Reachability-Testing* in Fig. 7 computes a race set for each receive event in a test sequence generated from model M during reachability testing. The race sets are used by procedure *GenerateVariants* to generate the variants of each collected sequence. Thus, the race set for each receive event in test sequence *s* is known when implementation CP is executed with test sequence *s*. We assume that these race sets are included in *s*. When test sequence *s* is used in step (b1) to collect sequence *s'* of CP, an *OpenList* and *race set* is computed for each receive event in *s'*, as described in Section 5.1.

Example 7.2: Fig. 12(a) shows a model M and Fig. 12(b) shows a feasible sequence *s* of M. The *OpenLists* of the receive events in sequence *s* are shown in the event descriptors as a list of annotations between brackets [...]. The only receive event with a non-empty race set is *r1*. Note that s_3 is not in

r_2*.race_set* because s_3 is not in the *OpenList* of r_2, and thus s_3 is not open at r_2. Fig. 12(c) shows implementation CP of model M. Fig. 12(d) shows the sequence *s'* exercised by implementation CP that corresponds to test sequence *s* of M in Fig. 12(b). In sequence *s'*, thread P_4 executes a selective wait statement with open receive-alternatives for ports *p1* and *p2*. The *OpenList* [*p1*,*p2*] for receive event *r2'* indicates that during execution, when the receive-alternative for *p1* was selected, port *p2* was also open. The race set for *r2'* contains send event *s3'* whose port *p2* is in the *OpenList* of *r2'*.



L1 p1
L2 p1
L3 p2
p1
L4
p2
L1
p1 (?,L4,p1,synch-send,p1)
L2
p1 (?,L4,p1,synch-send,p1)
L3
p2 (?,L4,p2,synch-send,p2)
L4
p1(?,L4,p1,synch-receive,p1)
p1(?,L4,p1,synch-receive,p1)
p2(?,L4,p2,synch-receive,p2)
Model M = (L1 | L2|L3 | L4) \ {p1,p2}.
L1
L2
L3
L4
OpenList
Race Set
s1
r1 (L1,L4,p1,synch-receive,p1,1,1)
[(?,L4,p1,synch-receive,p1)
?,L4,p1,synch-receive,p1)]
{s2}
s2
r2 (L2,L4,p1,synch-receive,p1,1,2)
[(?,L4,p1,synch-receive,p1)]
{ }
s3
r3 (L3,L4,p2,synch-receive,p2,1,3)
[(?,L4,p2,synch-receive,p2)]
{ }
P1
P2
P3
P4
OpenList
Race Set
s1'
r1' (P1,P4,p1,synch-receive,p1,1,1)
[p1]
{s2'}
s2'
r2' (P2,P4,p1,synch-receive,p1,1,2)
[p1,p2]
{s3'}
s3'
r3' (P3,P4,p2,synch-receive,p2,1,3)
[p2]
{ }

(a)

(b)

```
p1.receive();
select
  p1.receive(); p2.receive();
or
  p2.receive(); p1.receive();
end select
```

Implementation CP of model M

(c)

(d)

Figure 12. Comparing race sets.

In step (b2), the *race_sets* of receive event *r* in test sequence *s* and *r'* in sequence *s'* executed by implementation CP are compared as follows:

- If *r.race_set* $\subset$ *r'.race_set* then there is a feasible sequence of CP that is infeasible for M; thus, the verdict *fail* is assigned.
- If *r.race_set* $\supset$ *r'.race_set* then there is a sequence *t* in Feasible_M that is infeasible for CP; thus, the verdict *fail* is assigned. Note that this failure will also be detected when *t* is used to verify relation M $\leq_F$ CP, since *t* is feasible for M but not CP.
- If *r.race_set* $\approx$ *r'.race_set* then the verdict *pass* is assigned.

Example 7.3: In Fig. 12(d), *r2'.race_set* = {*s3'*} but in Fig. 12(b), *r2.race_set* = { }. Thus, there is a feasible sequence of CP that is infeasible for M, and the verdict *fail* is assigned.

Theorem 7.1. If procedure $\text{Test}_{\approx F}$ is performed and all the tests in Feasible_M *pass* then M $\approx_F$ CP.

Proof: See Section A.2 in the Appendix.

8. Using Reachability Testing to Verify M $\approx_F$ CP with Local Test Sequences

Each SR-sequence generated from model M during reachability testing can be used as a global test sequence for testing implementation CP. As reported in Section 9, reachability testing generates significantly fewer global (partially-ordered) test sequences than global test generation techniques that use an interleaving concurrency model. Still, the number of global test sequences generated by reachability testing may be very large. Using *local tests* may significantly further reduce the number of test sequences that must be executed by the implementation. In this section, we describe how reachability testing is used to generate local test sequences.

8.1 **Generating Local Test Sequences**

The following procedure uses algorithm *Reachability-Testing* to generate Constrained-Sequences(L_i) in Def. 4.2 for the ALTSs in acyclic model M.

Procedure *ProjectFeasibleSequences*:

1. For each feasible (partially-ordered) sequence *s* collected during reachability testing of M, project *s* onto L_i $1 \leq i \leq n$, to generate a complete local sequence s_{Li} for L_i and add s_{Li} to Constrained-Sequences(L_i). When projecting sequence *s*, event *e* in *s* is appended to s_{Li} if *e* is a send event and L_i is the sender, or *e* is a receive event and L_i is the receiver.

2. Generate all the non-empty, proper prefixes of s_{Li} and add them to Constrained-Sequences(L_i).

In the space-time diagram for sequence *s*, the local sequence s_{Li} for LTS L_i is simply the send and receive events in the vertical time-line for L_i.

Theorem 8.1: Procedure *ProjectFeasibleSequences* generates Constrained-Sequences(L_i), for every $i \in \{1..n\}$.

Proof: Algorithm *Reachability-Testing* generates every possible feasible, partially-ordered, SR-sequence of M and hence every possible feasible, constrained, local sequence of ALTS L_i.

We point out that a simple optimization of procedure *ProjectFeasibleSequences* is to remove step (2), which generates all the non-empty, proper prefixes of the local test sequence $\mathrm{s_{Li}}$ generated in step (1). These prefix sequences are members of Constrained-Sequences(L_i), but they are redundant and can be safely ignored during testing. For example, if local test sequence *a.b.c* is generated, it is not necessary to generate prefix sequences *a.b* and *a*. The reason being that if sequence *a.b.c* is feasible for the implementation thread, then sequences *a.b* and *a* must also be feasible, and there is no need to test the prefix sequences separately.

Recall that when global SR-sequence *s* is generated from model M during reachability testing, race sets are computed for the receive events in *s*. When *s* is projected to derive a local sequence for each L_i, the race set for receive event *r* in sequence *s* is retained by *r* in any local sequences that contains *r*. Another piece of information that must be retained by *r* is the timestamp for the send event that is synchronized with *r* in sequence *s*. Recall that timestamps are generated for each event during reachability testing in order to determine the happened-before relation among the events. In Section 8.2.1, we describe how timestamps for the send events are used during local testing.

Definition 8.1: Two local test sequences *s1* and *s2* in Constrained-Sequences(L_i) are equivalent if the i^{th}, i>0, events in these sequences are equivalent (Def. 3.16) and, for receive events, have identical race sets.

Two local sequences may have equivalent events, but there may be a receive event *r* that has a different race set in sequence *s1* than it has in *s2*. In this case, *s1* and *s2* are not considered to be duplicate sequences, and both sequences are included in Constrained-Sequences(L_i) and used for local testing.

8.2 A Test Procedure for Local Testing

A local-testing procedure for verifying the local implementation relations defined in Section 4 is shown in Fig. 13. Procedure $LocalTest_{\approx F}$ is similar to the global testing procedure in Fig 11, except that $LocalTest_{\approx F}$ is applied to each LTS-Thread pair (L_i, P_i); thus, $LocalTest_{\approx F}$ uses Constrained-Sequences(L_i) instead of Feasible$_{\mathrm{M}}$.

```
Procedure $LocalTest_{\approx F}$: For each mapped pair ($L_i$, $P_i$), $1 \le i \le n$:
(a) Generate Constrained-Sequences($L_i$).
(b) For each test sequence $s_{Li}$ in Constrained-Sequences($L_i$):
    (b1) Test $P_i$ with sequence $s_{Li}$ and inputs($s_{Li}$) If $P_i$ fails with $s_{Li}$, testing halts.
    (b2) Let $s'_{Pi}$ be the sequence executed by $P_i$ that corresponds to $s_{Li}$. For each
         receive event $r$ in $s_{Li}$ :
            Let $r'$ be the corresponding receive event executed in $s'_{Pi}$. If
            ($r.race\_set = r'.race\_set$) then assign the verdict pass; else assign fail.
            If $P_i$ fails with test sequence $s_{Li}$, then a failure has been detected and
            testing halts.
```

Figure 13. A local test procedure for M $\approx_F$ CP.

Procedure $LocalTest_{\approx F}$ has two parts. The first part is to generate the sequences in Constrained-Sequences(L_i) and determine whether the relation $L_i \le_F P_i$ holds. This first part of the procedure consists of steps (a) and (b1). A stateless procedure *ProjectFeasibleSequences* for generating Constrained-Sequences(L_i) in step (a) was presented in Section 8.1. In step (b1), thread P_i is executed with a test driver to determine the feasibility of local test sequence s_{Li}. The driver provides a conforming environment for sequence s_{Li} by supplying the send and receive events that match the events executed by P_i in local sequence s_{Li}. That is, whenever sequence s_{Li} has a synchronous or asynchronous, abstract send event *snd* for P_i to execute, then the corresponding concrete event executed by P_i is denoted as *snd'* and a concrete, synchronous or asynchronous receive event *rcv'* is executed by the driver. Likewise, whenever sequence s_{Li} has a synchronous or asynchronous, abstract receive event *rcv* for P_i to execute, then the corresponding concrete event executed by P_i is denoted as *rcv'*, and a concrete synchronous or asynchronous send event *snd'* is executed by the driver. Note that the execution of a single thread P_i interacting with a test driver will be deterministic. If sequence s_{Li} calls for P_i to execute a send or receive event, the test driver issues a single matching receive or send event, and no other events are executed by the driver unless and until thread P_i executes its expected event.

Procedure $LocalTest_{\approx F}$ can be classified as a gray-box technique – it requires access to the source code of the threads, so that the threads can be compiled with the test driver, but test sequences are generated from the specification model, not the implementation. Our local-testing technique can be implemented without the use of a deterministic testing tool, as the execution of the thread under test and the test driver is deterministic. Our global testing technique relies on testing tools that control inter-thread synchronization in order to force the implementation to deterministically execute global test sequences [Carver and Tai 1991; Tai et al. 1991; Tai and Carver 1996]. Gray-box testing is appropriate during earlier phases of the life-cycle, when implementation-level (i.e., the code, runtime, or operating system) observation and control is available. This allows bugs to be found early, when they are less costly to fix.

Timestamps are generated for the events executed by P_i and the events executed by the test driver. The timestamps are needed in order to compute the happened-before relations between these events. Note that the timestamp for a send event *snd'* that is executed by the test driver is not generated during the execution of test sequence s_{Li}. Instead, the timestamp for *snd'* is supplied by the receive event in s_{Li} that synchronizes with *snd*. As described in Section 8.1, the supplied timestamp is the timestamp that was generated for event *snd* in the sequence *s* that was projected to create s_{Li}. This timestamp reflects the happened-before relations between *snd* and the other events in *s* and thus ensures that the same relations are computed between *snd'* and the other events executed by the test driver and by thread P_i.

The test driver must supply concrete message objects for the send events that it executes. The sent objects are received by thread P_i when it executes the receive events in test sequence s_{Li}. To assist in this process, reachability testing can be applied to implementation CP to capture the message objects that are received by P_i. Captured message objects can be stored in a map structure that maps a message label to the associated message object of P_i. When the test driver needs to send a message to P_i with a given label, it can use the label to retrieve the appropriate message object from the map.

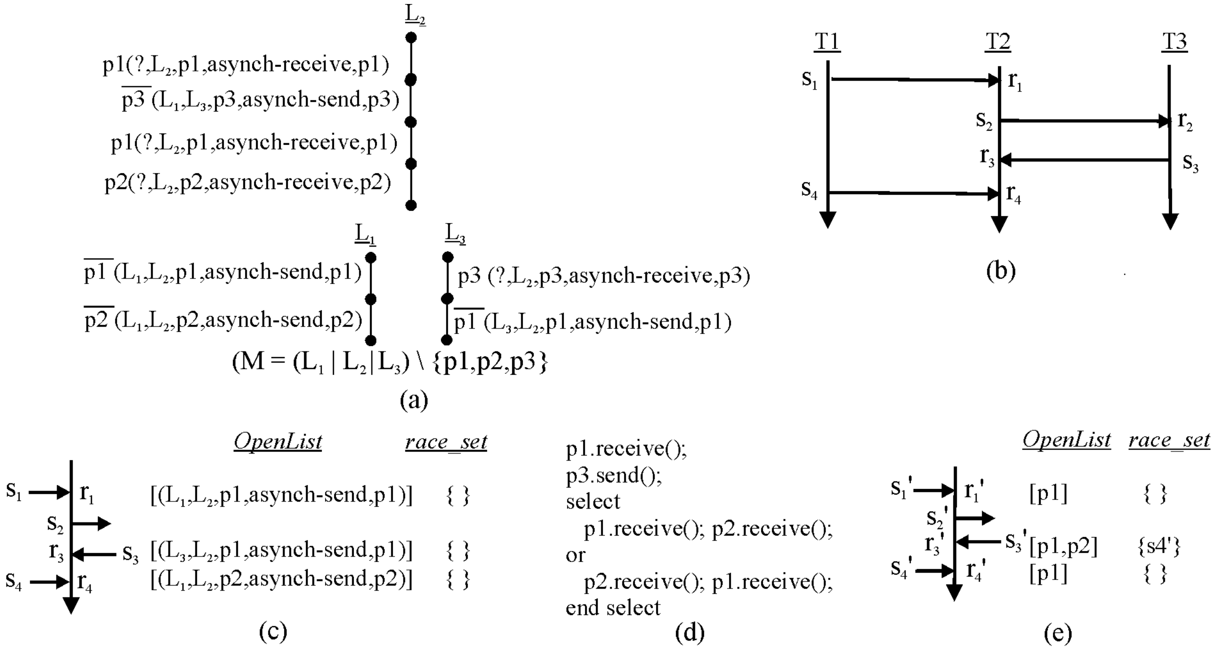
Note that it is not necessary for reachability testing that is used in this way to exhaustive; reachability testing of CP can stop when all or most of the message objects have been seen, or when a user-specified time limit is reached. If some message labels are not observed before reachability testing stops, then the user must supply the missing objects, possibly by modifying captured objects. The non-exhaustive version of reachability testing developed in [Lei et al. 2007] can be used to quickly collect message objects. Souza et al. [2011] show how to use static analysis to guide non-exhaustive reachability testing so that selected structural coverage criteria, such as executing all send events, are satisfied as early as possible. This technique can be applied with non-exhaustive reachability testing to speed up the collection of message objects from implementation CP. Note that it is not necessary to check whether CP issues the correct messages in the correct order during reachability testing. This will be checked during local testing. We simply capture the message object associated with each label. This can be considered as part of the process to verify that message objects are correctly labeled based on the values of the messages.

The second part of test procedure $\mathrm{LocalTest}_{\approx F}$ checks relation $P_i \leq_F L_i$. Step (b2) determines whether P_i can exercise any sequences that are not in Constrained-Sequences(L_i) when P_i is executed with inputs modeled by L_i. This is done just as it was done in step (b2) of procedure $Test_{\approx F}$ in Section 7.2 — by checking the termination behavior of $\mathrm{P_i}$, and comparing the race sets of the receive events executed by L_i and P_i. Recall that the information maintained for each receive event *r* in Constrained-Sequences(L_i) includes the race set of *r*. Below, we describe how to compute the race set of the corresponding receive event *r'* executed by P_i during local testing, and how to assign a test verdict based on a comparison of the race sets for *r* and *r'*.

8.2.1 Race Sets for Receive Events Executed by Thread $\mathrm{P_i}$

Let s_{Li} be a test sequence in Constrained-Sequences(L_i) and s'_{Pi} be the corresponding sequence exercised by thread $\mathrm{P_i}$. The race set of receive event *r'* in s'_{Pi} is computed using the timestamps for the send events executed by the test driver and the *OpenList* of *r'*, based on Proposition 1 in Section 5.1. This proposition uses the happened-before relation to identify racing send events executed concurrently with *r'*. As described above, the send events are executed by the test driver. The timestamps for these send events are obtained from the matching receive event *r* in s_{Li} and used to compute the happened-before relations.

Example 8.1: Fig. 14(a) shows a model M and a global sequence *s* of M (Fig. 14(b)). When *s* is projected to create local sequence s_{L2} in Fig. 14(c), the timestamps of send events *s1*, *s3*, and *s4* are stored with their corresponding receive events *r1*, *r3*, and *r4* in local sequence s_{L2}. These timestamps in s_{L2} will show that *s1* happened-before *s3* and that *s3* and *s4* are executed concurrently. (In sequence *s* in Fig. 14(b), if there is a path from event *e1* to event *e2* that follows the direction of the arrows, then *e1* happened-before *e2*.) Note also that whenever a receive event is executed in s_{L2} in Fig. 14(c), there is only one open port. Consequently, the race sets of all the receive events in s_{L2} are empty. Fig. 14(d) shows an implementation of thread P_2 and the local sequence s'_{P2} (Fig. 14(e)) that is exercised by P_2 when it is tested with local sequence s_{L2}. In sequence s'_{P2}, the timestamps used for send events *s1'*, *s3'*,


L₂
p1(?,L₂,p1,asynch-receive,p1)
p3 (L₁,L₃,p3,asynch-send,p3)
p1(?,L₂,p1,asynch-receive,p1)
p2(?,L₂,p2,asynch-receive,p2)
L₁
L₃
p1 (L₁,L₂,p1,asynch-send,p1)
p2 (L₁,L₂,p2,asynch-send,p2)
p3 (?,L₂,p3,asynch-receive,p3)
p1 (L₃,L₂,p1,asynch-send,p1)
(M = (L₁ | L₂|L₃) \ {p1,p2,p3}
(a)
T1
T2
T3
s₁
r₁
s₂
r₂
r₃
s₃
s₄
r₄
(b)
OpenList
race_set
[(L₁,L₂,p1,asynch-send,p1)] { }
[(L₃,L₂,p1,asynch-send,p1)] { }
[(L₁,L₂,p2,asynch-send,p2)] { }
(c)
(d)
s₁'
r₁'
s₂'
r₃'
s₃'
s₄'
r₄'
[p1] { }
[p1,p2] {s4'}
[p1] { }
(e)

```
p1.receive();
p3.send();
select
   p1.receive(); p2.receive();
or
   p2.receive(); p1.receive();
end select
```

Figure 14. Race sets of receive events in local sequences.

and *s4'* executed by the test driver are the timestamps these events had in *s*. Since the implementation of P_2 uses a select statement to choose between receive statements for ports *p1* and *p2*, the *OpenList* for *r3* contains both *p1* and *p2*. In s'_{P2}, *r1'.race_set = r4'.race_set* = { }, and *r3'.race_set* = {*s4'*}. Thus, the race sets of *r3* and *r3'* are different.

8.2.2 Comparing Race Sets of Receive Events in Local Sequences

Step (b2) in procedure Test$_{\approx F}$ compares the race sets of the receive events executed by L_i and P_i. Let s'_{Pi} be the sequence executed by P_i that corresponds to s_{Li}. For each receive event *r* in s_{Li} and the corresponding event *r'* in s'_{Pi}, if *r.race_set* ≈ *r'.race_set* then the verdict *pass* is assigned; otherwise, fail is assigned. Just as for non-local testing, if *r.race_set* ⊂ *r'.race_set* then there is a feasible sequence of P_i that is infeasible for L_i.

Theorem 8.2: If procedure *LocalTest$_{\approx F}$* is performed and all the tests in Constrained-Sequences(L_i) receive a verdict of *pass* then M $\approx_F$ CP.

Proof: See Section A.2 in the Appendix.

Note that when thread P_i is tested with the local sequences in Constrained-Sequences(L_i), the tests are used to verify that P_i will interact as intended with the other threads in the program. There is no circular reasoning used in this approach — the other threads are not assumed to be correct when testing P_i, and P_i is not assumed to be correct when the other threads are tested. The other threads may have faults that would prevent them from correctly interacting with P_i. These faults will be detected when the other threads are tested in turn with their local test sequences.

The sum of the sizes of Constrained-Sequences(L_i) over all L_i may be a small fraction of the number of feasible sequences of M. This is, however, not necessarily the case. For example, if ALTS L is a thread that interacts with all the other threads in the system, then each feasible sequence of the system might correspond to a different local sequence of L and no reduction will be achieved by using the local test sequences in Constrained-Sequences(L) instead of Feasible$_\mathrm{M}$. Such a result is reported in the case study in Section 9. The number of partially-ordered test sequences may still be drastically lower than the number of totally-ordered sequences.

9. Empirical Study

We conducted an empirical study on global and local test oracles using the testing framework described in Section 1. The study was performed on several versions of the following three models and their Java implementations, which were manually developed by the authors.

- *DP*: a solution to the dining philosophers problem with up to 9 philosopher threads. Hungry philosophers sit in a circle. In order to eat, a philosopher must pick up the two forks beside them – one is on their left and one is on their right. Each fork is shared with a neighboring philosopher. All the philosophers but one pick up their left fork first, while the "odd philosopher" picks up its right fork first. Each philosopher eats once.
- *TDME*: a token-based solution to the distributed mutual exclusion problem [Suzuki and Kasami 1985] with 3 user threads and 1 controller thread. User threads that wish to have exclusive access to a shared resource must obtain a special token from the controller thread. Each thread uses the shared resource one time.
- *DME*: a solution to the distributed mutual exclusion problem [Ricart and Agrawala 1981] in which processes communicate using asynchronous message passing. Each process contains three threads. A process that requires exclusive access to a shared resource must send requests to all the other processes and wait for all the other processes to reply. Requests are time stamped with logical clock values so that a winner can be chosen when more than one process makes a request. Each process uses the shared resource one time.

All the models and implementations were acyclic. Note that while TDME and DME solve the same problem, they have significantly different synchronization behavior. The DP and DME specifications were written in Lotos. Component ALTSs of the Lotos specification models were generated using the Lotos CADP toolset. Test sequences were generated using the *RichTest* stateless reachability tool and the test procedures described in Sections 6 - 8. We developed our own programs for generating and counting local and global test sequences from the ALTS models. The artifacts used in this empirical study can be found at [Carver and Lei, 2017].

Our objective here is to study the effectiveness of local tests for detecting violations of the implementation relations and to compare the number of test sequences generated for local testing to the number of sequences generated by other approaches.

Table I summarizes the results of test sequence generation. For models DP and DME, we use DP-i and DME-i to indicate that there were *i* philosophers and *i* processes, respectively, in the models. For each of the models, Table I shows:

(1) The number of states and transitions in the global ALTS model (column 2). Global ALTSs were generated using standard interleaving semantics and then minimized modulo strong equivalence in order to remove redundant sequences. The resulting global ALTSs contained no internal events.

(2) The number of global test sequences generated (columns 3 and 4). Global test sequences were generated using two different methods, which are described below.

(3) The number of local test sequences generated (column 5).

We attempted to build a global ALTS model of DME-4 using incremental and non-incremental reachability analysis; however, the ALTS model could not be built due to state explosion. We also tried to use the *distributor* tool in [Fernandez et al. 1996] to construct the global ALTS model of DME-4 on a compute cluster of 53 workstations, but there was still not enough available memory (approx. 100GB). The size of the global state space reported for DME-4 is the number of states and transitions that had been visited when memory was exhausted. The number of states remaining to be visited was still growing on most of the workstations.

Global test sequences were generated using two different methods. The first method reports the number of unique, *totally-ordered*, global sequences generated from the global ALTS models (column 3 of Table I). This is the number of global sequences that can be generated when concurrent events are modeled by enumerating their possible interleavings. We used a dynamic programming procedure to count the number of complete, totally-ordered sequences, from the initial state to a termination state, in

Model	Global ALTS model (states/ transitions)	#Totally-ordered sequences	#Partially-ordered sequences	#Local test sequences
TDME	192 / 348	67,894	30	33
DP-2	18 / 20	4	2	6
DP-3	76 / 126	238	6	9
DP-4	322 / 712	94,526	14	12
DP-5	1,364 / 3,770	108,549,484	30	15
DP-6	5,778 / 19,164	21.7×10^{10}	62	18
DP-7	24,476 / 94,710	9.1×10^{14}	126	21
DP-8	103,682 / 458,512	6.9×10^{18}	254	24
DP-9	439,204 / 2,185,074,	82.3×10^{21}	510	27
DME-3	367,733 / 1,403,821	72.2×10^{33}	4,032	315
DME-4	>264,471,486 / 1,199,776,000	$> 72.2 \times 10^{33}$	1,455,667,200	7,126

Table I. Test sequence generation.

the ALTS models. The first time a state *s* is encountered, the procedure computes the number of totally-ordered sequences from state *s* to a termination state. For each subsequent encounter of state *s*, the procedure does not re-compute the number of totally-ordered sequences from *s* to a termination state; rather, the procedure uses the number computed the first time state *s* was encountered.

Since the global ALTS model of DME-4 could not be constructed, we can only report that DME-4 has at least as many totally-ordered sequences as DME-3, although the actual number would be a great deal larger. The counting procedure counted, but did not generate, the totally-ordered sequences. Generating all of the totally-ordered sequences would be prohibitive for the larger models, as we could only generate on the order of 10^8 sequences per day.

The second method for counting sequences reports the number of global sequences generated by *RichTest*. This appears in Table I as the number of unique, *partially-ordered*, sequences (column 4). This number is considerably smaller than the number of totally-ordered sequences.

Local test sequences were generated using RichTest and the testing procedure described in Section 8. We point out that since every transition in a local test involves the thread under test, no two transitions in the same local test sequence are concurrent. This guarantees that no two local sequences differ only in the order of concurrent events, the same as for partially-ordered test sequences.

Table I (column 5) shows that the number of local test sequences was significantly less than the number of totally-ordered sequences generated from the global models for all but the smallest model, and was significantly less than the number of partially-ordered sequences generated for eight out of eleven of the models. Local tests were generated for TDME, the DP models, and DME-3 in under 4 seconds each. The time for DME-4 is discussed below. Executing the local tests against all of the implementation units took only a few seconds.

The number of local test sequences generated for model DP with P philosophers and P forks is always 3P. By way of comparison, a complete DP model has $2^P – 2$ unique partially-ordered sequences and considerable more totally-ordered sequences. For small numbers of philosophers, fewer partially-ordered sequences were generated than local sequences.

For the DME-3 model, a total of 315 local test sequences were generated for the nine implementation threads of the three processes. There are three threads per process. One thread sends and receives request and reply messages when it tries to enter the critical section, one thread processes request messages received from the other processes and sends them reply messages, and one thread is used to synchronize the first two threads. The global DME-3 model has 4,032 partially-ordered global sequences and 72.2 x 10^{33} totally-ordered, global sequences.

We measured the adequacy of the local test sequences generated for DME-3 by using mutation testing. Each mutation introduced a single change in the DME implementation that was intended to simulate a programming error. Examples of a mutation include replacing the relational operator “>=” with the operator “>”, replacing one variable with another from the same scope, and deleting an entire statement. A mutation creates a communication/synchronization error in the implementation such as making a given send or receive statement impossible to execute, causing a send or receive statement to be executed under the wrong conditions, or sending a message to the wrong process or with the wrong message value. Local test-sequences detect and reject mutants by causing the behavior of the original DME implementation to differ from the mutant. This is called *killing* the mutant. Mutants were generated using the traditional mutation operators and the Java-based mutation tool μJava [Ma et al. 2005]. Some of the mutants created were functionally equivalent to the original program, which means that they would always generate the same outputs and exercise the same feasible sequences as the original program. It is not possible to find test cases that can kill these mutants. Thus, functionally equivalent mutants were identified by manual inspection and deleted, which left 190 mutants. We then developed local test drivers and applied local testing to the nine implementation threads in DME-3. Each of the 190 DME mutants was killed by the local tests.

The global DME-4 model has 1,455,677,200 partially-ordered, global sequences but only 7,126 local test sequences were generated for the twelve implementation threads. The local sequences for the DME-4 model were generated by applying (distributed) reachability testing to DME-4 on a cluster of workstations [Carver and Lei 2010a]. During distributed reachability testing, different sequences are exercised concurrently by different workstations, achieving a nearly linear speedup over sequential reachability testing. Running distributed reachability testing on 175 of the cores available in a cluster of 53 multi-core workstations required 13.5 hours to exercise all 1,455,667,200 unique partially-ordered sequences of the DME-4 model, and generate the 7,126 local test sequences needed to verify relation M $\approx_F$ CP.

As we mentioned above, a global ALTS model could not be built for DME-4 due to state explosion. This prevented us from verifying the correctness of the DME-4 model using stateful verification techniques. Thus, during stateless reachability testing, we verified the following property of the DME-4 model: every process enters the critical section and there is never more than one process in a critical section at a time. This property was checked for each global sequence that was generated from the DME-4 model during reachability testing.

Note that distributed reachability testing can also be applied to the Java DME-4 implementation to generate all of the sequences allowed by the implementation. These sequences can be used to verify relation CP $\leq_F$ M. However, applying reachability testing to the implementation takes approximately 10 times longer to complete than applying reachability testing to the ALTS model. Fortunately, this is not needed since, based on Theorem 4.5 both M $\leq_F$ CP and CP $\leq_F$ M, and hence M $\approx_F$ CP, can be verified using the 7,126 local test sequences generated from the DME-4 model. We are not aware of any technique for directly generating local tests from an implementation to verify relation CP $\leq_F$ M.

For the TDME model, a total of 33 local test sequences were generated for the four implementation threads. The global TDME model has 30 partially-ordered, global sequences and 67,894 totally-ordered, global sequences. Thus, the number of local test sequences was lower than the number of totally-ordered, global sequences but slightly higher than the number of partially-ordered, global sequences. In TDME, most of the interactions are between the user threads and the controller thread, the result being that the number of local sequences of the controller thread is the same as the number of unique, partially-ordered, global sequences in the model. Since each of the three user threads has a single local test sequence, the total number of local tests sequences is 3 more than the number of partially-ordered global sequences.

For comparison, we applied the local testing technique presented in [Carver and Lei, 2013] and described in Section 10 to the four models. Briefly, this technique composes the component ALTSs in M to construct a reduced model that captures the intended interactions between a single component and the other components in the model. The reduced model is built by applying standard equivalence-based reductions algorithms to the ALTSs during compositions. The reduced model for a single component is typically much smaller than the global ALTS model for M.

The numbers of local tests derived using this stateful technique were the same as the numbers reported in Table I for TDME, the DP models, and DME-3. However, the stateful technique takes longer to generate local tests. For example, it takes close to 5 minutes to generate local tests for DME-3 using the stateful technique, but only 4 seconds using stateless reachability testing. The stateful technique was not able to generate thread interaction models for DME-4, due to state explosion. This was true even when incremental reachability analysis techniques were used on the compute cluster of 53 workstations. Distributed, stateless local testing avoids the high storage costs associated with interleaving concurrency models, while achieving linear speedup, which enabled it to check correctness properties and generate local tests from the DME-4 model.

Finally, we discuss the threats to the validity of our case study. The main threat to external validity is the degree to which the subject programs are representative of true practice. The programs studied were small in terms of lines of code, but they represent complex, classical synchronization patterns and they illustrate the range of test sequence reduction that can be achieved by local testing. The threat to external validity can be reduced by conducting experiments on more programs. The main threat to internal validity is the possibility that errors were made in generating and counting the test sequences. The partially-ordered sequences were generated and counted using the *RichTest* tool in [Lei and Carver 2006]. Local sequences were generated and counted by code that we developed, which was carefully tested. The results of local testing agreed with the results in [Carver and Lei, 2013], which were generated using different tools. The totally-ordered sequences were counted using a dynamic programming algorithm, whose implementation was carefully tested.

10. Related Work

In this section, we review related work on deriving test oracles from formal models of concurrent systems. We focus on models that are expressed as, or can be translated into, labeled transition systems.

We first note that test sequences can also be generated by analyzing an implementation's structure [Yang and Chung 1992; Yang et al. 1998] or its runtime behavior [Godefroid 1997; Kim et al. 1996]. Model-based and implementation-based testing are complementary approaches, in that certain faults may be detectable when using an implementation-based approach but not when using a model-based approach, and vice versa [Mouchawrab et al. 2011]. For example, the implementation-based reachability-testing tool in [Lei and Carver 2006] can exercise every path of a concurrent program with a given input, but it cannot directly detect “missing paths”, i.e., paths that are allowed by the specification but not allowed by the implementation.

We also note that there has been work in the area of local or compositional model checking [Flanagan and Vardi 1997]. The basic idea is to infer global properties of the whole system from the results of verifying individual components. This typically requires the user to provide an assumption about the interaction between the component being checked and the rest of the system. Our local testing technique generates test sequences from a model of the whole system. In addition, our test generation technique does not perform any model-checking on the implementation or the abstract model. Model checking is complementary to our work ─ the abstract model may be verified using model checking before the model is used to generate tests for the concrete implementation.

10.1 Conformance Testing

Most existing model-based test oracles for concurrent systems are based on a finite state machine (FSM) model such as an I/O automaton [Lynch and Tuttle 1989] or I/O state machine [Phalippou 1994], or they use an LTS model [Barr et al. 2015]. Since LTS models that have a finite set of states can be converted into FSM models and vice versa [Broy et al. 2005], oracles developed for one type of model can also be applied to the other. These oracles have been developed mainly in the context of *conformance testing* for protocol implementations. Protocols are often modeled as communicating *sender* and *receiver* components, each of which is implemented as a single process that is tested separately.

The implementation under test (IUT) conforms to specification model M if and only if the output responses of the IUT and M to each test sequence coincide [Dorafeeva et al. 2010; Lee and Yannakakis 1996]. When M is a *single* deterministic finite state machine, a test is performed by providing the IUT with a sequence of inputs selected from M, called a test sequence. Test sequence generation methods for finite state machines are compared in [Endo and Simao 2012]. These methods differ on the assumptions they make about the model and the implementation [Dorafeeva et al. 2010; Lee and Yannakakis 1996].

The “oracle” part of protocol conformance testing can be viewed as checking, e.g., that the output produced by the IUT on a given transition T matches that predicted by M, or that the state reached in the IUT after transition T corresponds to the state prescribed by M [Baresi and Young 2001].

Another approach to conformance testing is to define a required implementation relation between the IUT and its model. Examples of such relations include ioco [Tretmans 1999], conf, and testing equivalence [Pitt and Freestone 1990], and there are many others. The implementation relation captures the ways in which the states and transitions of the IUT are allowed to differ from the states and transitions of the model. Tests are generated to detect differences that are not allowed by the implementation relation. The ioco relation is described below.

In general, conformance-testing techniques are *black-box* techniques. Black-box techniques encounter problems when they are applied to non-deterministic systems. During black-box testing, non-deterministic implementation events are unobservable and uncontrollable, which produces inconclusive test results. As we mentioned in Section 8.2, our global and local testing techniques are a form of *gray-box testing*. Tests are selected from the specification, and implementation-level observation and control is used to address the problems caused by non-determinism.

Another problem with conformance testing techniques is that the composite FSM and LTS models are often based on the *interleaving model* of concurrency, which was described in Section 1. This creates an explosion in the number of modeled states, making it impossible to store the entire state space in memory. Along a different line, several stateful techniques [Jard 2002; Ponce de Leon et al. 2013; Ulrich and Chanson 1995; Ulrich and König 1997] have been developed for generating global test sequences from true-concurrency state-space models. In such models, a global test sequence is a partial order of transitions in which concurrent transitions are left unordered. True-concurrency models can significantly reduce the number of test sequences, but they may still be large. Our global, stateless, test generation technique also generates partially-ordered sequences, but it does not require any state-space models to be built. In addition, our stateless technique can be used to generate local sequences for individual threads. As the case study shows, the total number of local sequences may be significantly smaller than the total number of global sequences, even when an interleaving-free approach is used to generate global sequences.

10.2 Compositional Conformance Testing

Several compositional conformance-testing techniques have been developed. Testing is performed individually on implementation components with test sequences generated from the corresponding individual specification components. Van der Bijl et al. [2004] showed how to perform compositional testing based on the *ioco* implementation relation. (Roughly, an implementation and its specification are ioco conformant if the implementation can never produce an output that cannot be produced by its specification after the same trace.) They determined sufficient conditions under which the *ioco* conformance of each of *two* LTS component implementations to their respective LTS component specification models leads automatically, without any additional testing, to the *ioco* conformance of the system implementation to the system specification. The condition that is relevent to this paper is that both component LTS specification models are *input enabled*. An LTS is input enabled if each state specifies a response for every possible modeled input. It is also assumed that each implementation component is input enabled.

Gotzhein and Khendek [2006] presented a compositional technique for testing protocol implementations that can be modeled as the composition of *two* input enabled, deterministic FSMs. Each of the two implementation components is tested separately using traditional black box testing methods. When the two components pass their local tests, test sequences are generated to detect composition faults in the code used to connect the input/output queues of the components. The test sequences for the connection code are generated without building the global FSM, and do not repeat the local test sequences already performed on the two components. The local-testing technique that we presented in this paper can be applied to models with more than two components, and if the local tests are passed, no separate integration tests are required.

An important issue with local and compositional testing is whether a component model specifies the exact set of inputs that can be received in a particular state. The compositional techniques in [Gotzhein and Khendek 2006] and [Van der Bijl et. al. 2004], as described above, require each individual component model to be input enabled – each state must specify a response for every possible modeled input. One problem with this requirement is that some inputs may be impossible in certain states, and it is not clear what response should be specified for an impossible input (see Example 4.2 in Section 4.2) In addition, manually identifying the impossible inputs of a state in a component is difficult when the possible inputs depend on complex interactions among two or more other components.

Another problem with this requirement occurs when some inputs are available, i.e., messages have been sent, and their availability is not an error, but receiving and responding to these inputs is not allowed in a certain state, e.g., a request for a resource has been sent, but the request is not allowed to be received when no resources are available. It is not clear how to specify in an input enabled model that certain available inputs are not allowed to be received or responded to. Our local-testing technique does not require LTS models to be input enabled, nor does it require implementation threads to have all inputs enabled in all states. The LTS model of an individual thread is permitted to contain states that allow inputs that are impossible in a global context, or that disallow inputs that are available. Impossible inputs are not a problem in our framework; the process used to generate tests implicitly identifies impossible inputs and prevents them from being included in the local test sequences that are generated.

The stateful local oracles presented by the current authors in [Carver and Lei, 2013] are constructed by building a thread interaction model for each thread in the implementation and deriving test sequences from these models. A thread interaction model is an annotated labeled transition system (ALTS) that captures the intended interactions between a single LTS L and the other LTSs in system model M. The number of states and transitions in a thread interaction model are typically much smaller than those in a global, stateful representation of model M. A thread interaction model for L is constructed using a series of equivalence-based reductions in an incremental manner, i.e., without building the global state space for M. Some empirical results about this process were presented in Section 9.

Although thread interaction models are typically very small, they are stateful models and cannot be constructed in cases where intermediate state explosion causes the reduction process to fail. In this paper, neither a stateful, global model of M nor a reduced, stateful model of M is constructed. Instead, paths of transitions through individual component models are analyzed to generate a sequence of synchronized send and receive transitions (SR-sequence) through model M. In addition, while the technique in [Carver and Lei 2013] only considers the relation $M \leq_F CP$, the technique in this paper can be used, with the same number of test sequences, to verify $M \leq_F CP$ and the stronger relation $M \approx_F CP$, using either global or local test sequences.

A local, stateful, approach to model inference and implementation testing is presented in [Groz et al. 2008]. In this approach, the system is executed with selected test inputs and the resulting execution traces observed for the system are used to infer FSM component models. Static, stateful reachability analysis is performed on the inferred component models to explore all of the possible execution interleavings and detect potential system errors such as unspecified receptions, livelocks, and races. Note that stateful reachability analysis is not feasible when the number of states is too large. The empirical results in Section 9 showed that our stateless approach might succeed in cases where stateful approaches exhaust memory.

Symbolic execution [King 1975] can be used to derive a symbolic execution tree, which describes all of the possible system executions in a symbolic way. Symbolic execution has also been used for component-based testing [Faivre et al. 2007; Kanso et al. 2012]. The focus is to prevent the generation of test sequences that are allowed by a component model, but that are not feasible in the context of the whole system. The technique in [Kanso et al. 2012] supports the compositional technique presented in [Van der Bijl et al. 2004] and described above, but using relation *cioco*, which is a slight adaption of the *ioco* relation. In addition, the specification models are not required to be input enabled.

A work closely related to ours is [Koppol et al. 2002], which presents a gray-box testing technique for generating test sequences from reduced ALTS models. A reduced state space for an ALTS model M is generated using incremental reachability analysis and a special ALTS reduction algorithm that stores information about the paths that are pruned from the unreduced state space of M. Each test sequence generated from a reduced ALTS corresponds to a complete path through the unreduced state space of the model. The generated test sequences can thus be used for global testing of the complete system. This is in contrast to our approach in which test sequences can also be generated for local testing of a single thread.

Also presented in [Koppol et al. 2002] are several local coverage criteria. Test sequences that satisfy these criteria can guarantee a level of coverage for the global model without ever having to build the global ALTS of the model. Satisfying these local coverage criteria verifies neither the implementation relation $M \leq CP$ nor the relation $CP \leq M$. However, these local coverage criteria could be used with our local testing technique when verifying the implementation relations requires too many local sequences to be executed − local tests can be executed while the local coverage criteria are measured to determine when to stop testing.

11. Conclusion

We presented a stateless technique for generating global and local test oracles for concurrent systems that are modeled as annotated labeled transition systems. Correctness is defined in terms of an implementation relation that is expected to hold between a model M of the system and its implementation CP. The implementation relation used in this paper is that M and CP allow the same sequences $M \approx CP$. We showed that this relation can be checked by adding a small amount of information to the test sequences that are used to verify the weaker relation $M \leq CP$.

A global test oracle can be used to verify a global implementation relation using global test sequences. Local implementation relations were also defined. Local relations exist between the individual component models and their implementation threads. We showed how to verify a global implementation relation by using local test oracles to separately verify each local relation. The novelty of our local test oracles is that the individual implementation threads are tested separately, without testing the system as a whole. This may enable a large reduction in the number of program executions that are required during testing. In addition, this localizes errors to a single implementation thread when a local test fails. Empirical studies confirmed that non-local testing with partially-ordered test sequences requires significantly fewer implementation executions than interleaving-based, global testing techniques, and that local testing may require significantly fewer implementation executions than global testing.

Our global and local test oracles are generated using the stateless reachability testing algorithm in [Lei and Carver 2006]. This is the first time we are aware of that a stateless, interleaving-free search technique has been applied to LTS models for generating test oracles. Implementation relations can be verified for models that are acyclic and closed. The test oracles do not require any state space models to be built or states to be stored. Thus, test oracles generated by our technique typically require less memory than oracles generated using stateful techniques. However, since states are not stored, it is impossible to recognize states that have already been visited during a stateless search, which may increase the time needed for generating test sequences.

Models for large, complex programs may admit a very large number of test sequences. Hence, a general limitation of our technique is that the test sequences required to verify an implementation relation might take too much time to generate and execute. If this is the case, then stateless reachability testing can be used to generate test sequences from the model while structural coverage criteria are measured against the implementation to determine when to stop testing [Souza et al. 2015; Zhu 1996].

We are currently working on an extension to our stateless reachability-testing algorithm that would allow all of the local test sequences of a model to be generated without generating all of the global, partially-ordered sequences of the model. This could significantly reduce the effort required for using stateless reachability testing to verify local implementation relations.

References

AMMANN, P., OFFUTT, J., 2008. *Introduction to software testing*. Cambridge University Press.

BARESI, L. AND YOUNG, M., 2001. Test oracles. Univ. of Oregon, Dept. Comput. Inform. Sci., Eugene, OR, SA. Tech. Rep. CIS-TR-01-02. [Online]. Available: http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

BARR, E., HARMAN, M., McMINN, P., SHABAZ, M., AND YOO, S., 2015. The oracle problem in software testing: a survey. IEEE Trans. Softw. Eng., 41(5):507-525.

BERNSTEIN, A., 1966. Analysis of programs for parallel processing. IEEE Transactions on Electron. Comput. 15(5), pp.757 – 763.

BOLOGNESI, T., AND BRINKSMA, E., 1987. Introduction to the ISO specification language LOTOS. Comput. Networks ISDN, Volume 14, Issue 1, 1987, pp. 25–59.

BRINKSMA, E., 1988. A theory for the derivation of tests. in: S. Aggarwal, K. Sabnani, eds., Protocol Specification, Testing and Verification, VIII, pp. 63-74.

BRINKSMA, E., SCOLLO, G., 1986. Formal notions of implementation and conformance in LOTOS. Rept. No. INF-86-13, Twente University of Technology, Department of Informatics, Enschede, The Netherlands.

BRINKSMA, E., SCOLLO, G., AND STEENBERGEN, C., 1987. Process specification, their implementations and their tests. in: G.v. Bochmann, B. Sarikaya, eds., Protocol Specification, Testing and Verification, VI, pp. 349-360.

BROY, M., JONSSON, B., KATOEN, J. P., LEUCKER, M., AND PRETSCHNER, A., ed., 2005. Model-based testing of reactive systems. Lecture Notes in Computer Science 3472. Springer-Verlag.

CARVER, R., 1996. Testing abstract distributed programs and their implementations. J. Syst. Softw., Special Issue on Software Engineering for Distributed Computing, June 1996.

CARVER, R., AND LEI, Y., 2010a. Distributed reachability testing. Concurrency Computat., Pract. Exper., Volume 22, Issue 18, pp. 2445-2466.

CARVER, R., AND LEI, Y., 2010b, A Class Library for Implementing, Testing, and Debugging Concurrent Programs. Int. J. Softw. Tools Technol. Transf.: Vol.12, No. 1 (2010), Page 69-88.

CARVER, R., AND LEI, Y., 2013. A modular approach to model-based testing of concurrent programs. Proc. International Conference on Multicore Software Engineering, Performance, and Tools (MUSEPAT), August.

CARVER, R., AND LEI, Y., 2017. Artifacts and programs for the case study, http://barbie.uta.edu/wp-content/uploads/2017/07/STVRFiles.zip.

CARVER, R., AND TAI, K.C., 1991. Replay and testing for concurrent programs. IEEE Software, pp. 66-74.

CARVER, R., AND TAI, K.C., 2006. Modern multithreading: implementing, testing, and debugging multithreaded Java and C++ Pthreads/Win32 programs. Wiley. http://www.cs.gmu.edu/~rcarver/ModernMultithreading.

CHEN, J., AND CARVER, R., 1996. Selecting and mapping test sequences from formal specifications of concurrent programs. Proc. of the High-Assurance Systems Engineering Workshop, October pp. 112-119.

CHUNG, I.S., KIM, B.M., AND KIM, H.S., 2001. A new approach to deterministic execution testing for concurrent programs. IEICE Trans. Inf. Syst. Vol. E84-D, No.12, pp. 1756-1766.

COLBY, C., GODEFROID, P., and JAGADEESAN, L. J., 1998. Automatically closing open reactive programs. Proc.1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 345-357.

CYPHER, R. and LEU, E., 1994. The semantics of blocking and non-blocking send and receive primitives. Proc. Eighth International Parallel Processing Symposium, pp. 729-735.

DORAFEEVA R., EL-FAKIH K., MAAG, S., CAVALLI, A., and YEVTUSHENKO, N., 2010. FSM-based conformance testing methods: A survey annotated with experimental evaluation. Inf. Softw. Technol. 01/2010; 52:1286-1297.

ENDO, A.T., AND SIMAO, A., 2012. Experimental comparison of test case generation methods for finite state machines. Proc. IEEE Fifth Int. Conference on Software Testing, Verification and Validation, pp. 549 – 558.

FAIVRA, A., GASTON, C., AND Le GAIL, P., 2007. Symbolic model based testing for component oriented systems. Testing of Software and Communicating Systems. Lecture Notes in Computer Science Volume 4581, pp. 90-106.

FEIJS, L.M.G., GOGA, N., MAUW, S., TRETMANS, J., 2002.Test selection, trace distance and heuristics. Proc. IFIP 14th Int. Conference on Testing Communicating Systems - TestCom, pp. 267-282.

FERNANDEZ, J., GARAVEL, H., KERBRAT, A., MATEESCU, R., MOUNIER, L., AND SIGHIREANU, M., 1996. CADP: A protocol validation and verification toolbox. Proc. 8th Conference on Computer-Aided Verification, pp. 437-440.

FIDGE, C. J., 1988. Timestamps in message-passing systems that preserve the partial ordering. Proc. 11th Australian Computer Science Conference (ACSC'88), pp. 56–66.

FLANAGAN C., and M. VARDI, 1997, Thread modular model checking. Proc. COMPOS'97, Lecture Notes in Computer Science, Vol. 1536, pp. 381 – 401.

GODEFROID, P., 1997. Model checking for programming languages using verisoft. Proc. 24th ACM Symposium on Principles of Programming Languages, pages 174-186, Paris.

GOTZHEIN, R., AND KHENDEK, F., 2006. Compositional testing of communication systems. TestCom 2006, LNCS 3964, pp. 227-244.

GROZ, R., LI, K., PETRENKO, A., SHAHBAZ, M., 2008. Modular system verification by inference, testing and reachability analysis. Testing of Software and Communicating Systems, Lecture Notes in Computer Science Volume 5047, pp. 216-233.

HONE, P., YASSINE, M., and Sofiène, T., 2002. Environment synthesis for compositional model checking. pp. 70-75, 2002.

HUGHES, G., and BULTAN, T., 2008. Interface grammars for modular software model checking. IEEE Trans. Softw. Eng., vol. 34, no. 5, pp. 614-632.

Information Technology, 1991. Open systems interconnection, conformance testing methodology and framework. International Standard IS-9646. ISO, Geneve.

IOUSTINOVA N., SIDOROVA, N., and STEFFEN, M., 2002. Closing open sdl-systems for model checking with DTSpin. LNCS Volume 2391, pp. 157-177.

JARD, C., 2002. Principles of distributed test synthesis based on true-concurrency models source. Proc. IFIP 14th International Conference on Testing Communicating Systems XIV, pp. 301-316.

KANSO, B., AIGUIER, M., BOULANGER, F., AND GASTON, C., 2012. Testing of component-based systems. Proc. Software Engineering Conference (APSEC), pp. 300-305.

KIM, M.C., CHANSON, S.T., KANG, S.W., AND SHIN, J.W., 1996. An approach for testing asynchronous communicating systems. 9th Int'l Workshop on Testing of Communicating Systems; Darmstadt, Germany.

King, J.-C., 1975. A new approach to program testing. Proc. of the Int. Conference on Reliable software, 21-23:228–233.

KOPPOL, P.V., CARVER, R.H., AND TAI, K.C., 2002. Incremental integration testing of concurrent programs. IEEE Trans. Softw.Eng., Vol. 28, No. 6.

LAMPORT, L., 1978. Time, clocks, and the ordering of events in a distributed. system. Comm. ACM, 21, 7 (July), pp. 558-565.

LEE, D., and YANNAKAKIS, M., 1996. Principles and methods of testing finite state machines-a survey. Proc. of the IEEE, Vol. 84, Issue 8, pp. 1090 – 1123.

LEI, Y., AND CARVER, R.H., 2006. Reachability testing of concurrent programs. IEEE Trans.Softw. Eng., Volume 32, No. 6, pp. 382-403.

LEI, Y., CARVER, R.H., KACKER, R. AND KUNG, D, 2007. A combinatorial testing strategy for concurrent programs. J. Softw. Test. Verif. Rel., Vol.17, Issue 4, pp. 207-225.

LYNCH N.A., AND TUTTLE, M.R., 1989. An introduction to Input/Output automata. CWI Quarterly, 2(3):219–246.

MA, Y.-S., OFFUTT, J, AND KWON, Y.-R., 2005, μJava: an automated class mutation system. J. Softw. Test. Verif. Rel., 15(2):97-133.

MATTERN, F., 1988. Virtual time and global states of distributed systems. Proc. Workshop on Parallel and Distributed Algorithms, pp. 215–226.

MILNER, R., 1989. *Communication and concurrency*. Prentice-Hall.

MOUCHAWRAB, S., BRIAND, L., LABICHE, Y., DI PENTA, M., 2011. Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments. IEEE Trans. Softw. Eng., 37(2):161-187.

PARIZEK, P., and PLASIL, F., 2007. Specification and generation of environment for model checking of software components. Proc. Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2006), ENTCS, Vol. 176, Issue 2, pp. 143-154.

PHALIPPOU, M., 1994. Executable testers. In Omar Rafiq, editor, Proc. 6th International Workshop on Protocol Test Systems (IWPTS 1993), volume C-19 of *IFIP Transactions*, pages 35–50, Pau, France.

PITT, D.H., AND FREESTONE, D., 1990. The derivation of conformance tests from LOTOS specifications. IEEE Trans. Softw. Eng., 16(12):1337–1343.

PONCE de LEON, H., HAAR, S., and LONGUER, D., 2013. Unfolding-based test selection for concurrent conformance. Testing Software and Systems, Lecture Notes in Computer Science, Vol. 8254, pp. 98-113.

RICART, G., AND AGRAWALA, A.K., 1981. An optimal algorithm for mutual exclusion in computer networks. Comm. ACM, 24, 1 (January), pp. 9-17.

SOUZA, SIMONE, SOUZA, PAULO, MACHADO, MARIO, CAMILLO, MARIO, SIMAO, ADENILSO AND ZALUSKA, Ed, 2011. Using coverage and reachability testing to improve concurrent program testing quality. Proc. 23rd Int. Conference on Software Engineering and Knowledge Engineering, pp. 207-212.

SOUZA, S. R. S., SOUZA, P. S. L., BRITO, M. A. S., SIMAO, A. S., AND ZALUSKA, E. J., 2015. Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. Software Testing Verification & Reliability, Vol**.** 25, Issue 3, pp. 310-332.

SUZUKI, I., AND KASAMI, T., 1985. A distributed mutual exclusion algorithm. ACM Trans. Comp. Syst., 3(4): 344-349.

TAI, K.C., 1985. On testing concurrent programs. Proc. COMPSAC 85, pp. 310-317.

TAI, K.C., 1997. Race analysis of traces of asynchronous message-passing programs. Proc.17th International. Conference on Distributed Computing Systems, pp. 261-268.

TAI, K.C., AND CARVER, R.H., 1996. Testing of distributed programs. Chapter 33 of *Handbook of Parallel and Distributed Computing*, ed. by A. Zoyama, McGraw-Hill, pp. 955-978.

TAI, K.C., CARVER, R.H., AND OBAID, E., 1991. Debugging concurrent Ada programs by deterministic execution. IEEE Trans. Softw. Eng., 17(1):45-63.

TAYLOR, R.N., 1983. A general-purpose algorithm for analyzing concurrent programs. Comm. ACM, 26(5), pp. 362-376.

TKACHUK, O., AND DWYER, M.B., 2003. Automated environment generation for software model checking. Proc.18th International Conference on Automated Software Engineering, pp. 116—129.

TRETMANS, J., 1999. Testing concurrent systems: a formal approach. Lecture Notes in Computer Science; Vol. 1664, Proc. 10th International Conference on Concurrency Theory, pp. 46 - 65.

TRETMANS, J., AND BRINKSMA, E., 2003. TorX: Automated model-based testing. Proc. First European Conference on Model-Driven Software Engineering, pp. 31-43.

ULRICH, A., CHANSON, S., 1995. An approach to testing distributed software systems. Proc. Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, pp. 121 – 136.

ULRICH, A., AND KÖNIG, H., 1997. Specification-based testing of concurrent systems. Formal Description Techniques and Protocol Specification, Testing and Verification, 17. T. Mizuno, et al. (Eds.).

VAN DER BIJL, M., RENSINK, A., AND TRETMANS J., 2004 Compositional testing with ioco. FATES 2003, LNCS 2931, pp. 86-100.

VISSERS, C, SCOLLO, G., AND VAN SINDEREN, M., 1988. Architecture and specification style in formal descriptions of distributed systems. (Invited) In: Proc. IFIP WG6.1 Eighth International Symposium on Protocol Specification, Testing, and Verification, pp. 189-204.

YANG, C., SOUTER, A.L., AND POLLOCK, L.L., 1998. All-du-path coverage for parallel programs. International Symposium on Software Testing and Analysis, pp. 153-162.

YANG, R.D. AND CHUNG, C.G., 1992. A path analysis approach to concurrent program testing. Information and Software Technology, 34(1):43-56.

ZAVE, P., 1984. The operational versus the conventional approach to software development. Comm. ACM, 27, 2 (February) pp. 104 - 118.

ZHU, H. 1996. A formal analysis of the subsume relation between software test adequacy criteria. IEEE Trans. Softw. Eng. 22(4):248–255.

Appendix

A.1 Definitions and Proofs for Section 4

Theorem 4.1: Let $Q = (s_{L1}, s_{L2}, ..., s_{Lm})$ be a feasible, partially-ordered SR-sequence of model M. Sequence Q is feasible for implementation CP iff for every $i \in \{1..n\}$ constrained local sequence s_{Li} is feasible for thread P_i.

Proof: Recall that when the feasibility of local sequence s_{Li} for thread P_i is determined, synchronizations between P_i and the other threads in CP do not actually occur. Instead, a conforming test environment simulates P_i*'s* environment in M by supplying the send and receive events that match the (annotated) events executed by P_i in local sequence s_{Li}. In addition, local sequence s_{Li} of ALTS L_i is a local sequence of global sequence Q that is *feasible* for M. Thus, the constrained local sequences of L_i capture the constraints that are imposed on L_i*'s* behavior by the other ALTSs in M.

For the only-if part, if the threads in CP are able to execute feasible sequence *s* of M, which includes all the events in all the local sequences of *s* and the synchronizations between threads as specified by the events of *s*, then thread P_i is also able to exercise the events in local sequence s_{Li} when a conforming test environment supplies any matching synchronizations that are needed by these local events according to the synchronizations in *s*.

For the if part, the question is whether the specific thread synchronizations in sequence *s* must be feasible if all the local sequences s_{Li} of *s* are feasible. By way of contradiction, assume s_{Li} is feasible for thread P_i, $1 \leq i \leq n$ but sequence *s* is not feasible for CP. Then there is an event *e* that is one of the (possibly many) events that can be the first event in sequence *s*. An event *e* is one of the first infeasible events in sequence *s* if no event in *s* that happened- before *e* is infeasible. Intuitively, an event e_1 *happened-before* another event e_2 if e_1 can potentially affect e_2 [Lamport 1978].The possible first events are executed concurrently. There are three cases in which event *e* in sequence *s* can be infeasible:

Case 1: Event *e* is an asynchronous send event executed by thread *P*: Since *e* is a non-blocking send event, the only way for *e* to be infeasible is for thread *P* to be unable to execute *e*, but this contradicts the assumption that local sequence s_P is feasible.

Case 2: Event *e* is an asynchronous receive event executed by thread *P*, where *C* is the thread executing the asynchronous send *e'* synchronized with *e*. Note that since *e* can be synchronized with *e'*, events *e* and *e'* must be concurrent events, which means that both *e* and *e'* are possibly one of the first infeasible events in *s*. Case 1 shows that send event *e'* cannot be infeasible in *s*. In order for receive event *e* to be infeasible in *s*, at least one of the following must be true:

(a) *P* is unable to execute receive event *e*, but this contradicts the assumption that s_P is feasible.

(b) P can execute receive event e, and C can execute send event e', but e' and e cannot synchronize with each other. However, this contradicts the assumption that sequence s is feasible for M, as e' and e must be synchronizable (i.e., have matching ports and labels) in order for s to be feasible in M.

Case 3: Event e is an *synchronous-synchronization* event, where C is the thread executing the synchronous send e_s for e and U is the thread executing the synchronous receive e_r for e. In order for e to be infeasible, at least one of the following must be true:

(a) Thread C is unable to execute e_s because thread C cannot execute a send event. But this contradicts the assumption that local sequence s_C is feasible.

(b) Thread U is unable to execute e_r because thread U cannot execute a receive event. But this contradicts the assumption that local sequence s_U is feasible.

(c) Thread C can execute send event e_s and thread U can execute receive event e_r, but e_s and e_r cannot synchronize with each other. But this contradicts the assumption that sequence s is feasible for M, as e_s and e_r must be synchronizable (i.e., have matching ports and labels) in order for s to be feasible in M. □

Theorem 4.2: : M $\leq_F$ CP iff for every $i \in \{1..n\}$ $L_i \leq_F P_i$.

Proof: For the only-if part, Theorem 4.1 says that if a feasible sequence $Q = (s_{L1}, s_{L2}, ..., s_{Lm})$ of M is feasible for CP, then local sequences s_{Li} of Q are feasible for the individual threads P_i of CP, $1 \leq i \leq n$. It follows directly from this that if all the feasible sequences of M are feasible for CP, then all of the constrained local sequences of L_i, $1 \leq i \leq n$, are also feasible for the individual threads P_i of CP.

For the if part, assume relation $L_i \leq_F P_i$, $1 \leq i \leq n$, holds but relation M $\leq_F$ CP does not. Then there is an event e that is one of the (possibly many) events that can be the first event in some feasible sequence $Q = (s_{L1}, s_{L2}, ..., s_{Lm})$ of M that is not feasible for CP. (An event e is one of the first infeasible events in Q if no event in Q that happened-before [Lamport 1978] e is infeasible. The possible first events are executed concurrently.) Assume that e is executed by ALTS L_j. Sequence s_{Lj} is a local sequence of L_j that is not feasible for P_i due to e, but we are assuming $L_j \leq_F P_j$. This is a contradiction that proves the theorem. □

Definition 4.7: A feasible, local SR-sequence t_{Pi} of thread P_i is *constrained with respect to program CP* if there exists a feasible, partially-ordered SR-sequence $Q = (s_{P1}, s_{P2}, ..., s_{Pm})$ of CP such that $t_{Li} = s_{Li}$. The set of constrained sequences of P_i with respect to program CP is denoted *Constrained-Sequences(P_i,*M*)*, or just *Constrained-Sequences(P_i)* when CP is understood.

Definition 4.8: Let sequence t_{Li} = (*sending thread*$_1$, *receiving thread*$_1$, *port*$_1$, *op*$_1$, *label*$_1$, j_1, *1*), (*sending thread*$_2$, *receiving thread*$_2$, *port*$_2$, *op*$_2$, *label*$_2$, j_2, *2*), ... , be a local sequence in Constrained-Sequences(P_i,CP). Local sequence t_{Pi} is *feasible for* L_i if L_i has a sequence of transitions $s_1 \xrightarrow{a_1}_{Ri} s_2 \xrightarrow{a_2}_{Ri} s_3 \ldots, s_i \in Q_i$, where state s_1 is the start state $q0_i$ of L_i, and for event (*sending thread*$_k$, *receiving thread*$_k$, p_k, op_k, e_k, j_k, k) and transition $s_k \xrightarrow{a_k}_{Ri} s_{k+1}$, k>0, of L_i, one of the following conditions is true:

- op_k is a send event and annotation a_k = (*sending thread*$_k$, *receiving thread*$_k$, p_k, op_k, e_k),
- op_k is a receive event and annotation a_k = (*?*, *receiving thread*$_k$, p_k, op_k, e_k);

otherwise, t_{Pi} is *infeasible* for L_i.

Theorem 4.3: Let $Q = (s_{P1}, s_{P2}, \ldots, s_{Pm})$ be a feasible, partially-ordered SR-sequence of CP. Then sequence Q is feasible for M iff, for every $i \in \{1..n\}$ constrained local sequence s_{Pi} is feasible for ALTS L_i.

Proof: Sequence Q (s_{Pi}) is *feasible for* M *(*L_i) if Q (s_{Pi}) corresponds to a sequence of synchronized steps (transitions) through M (L_i). For the only-if part, if the LTSs in M can be synchronized to create a sequence of synchronized steps through M that corresponds to $Q = (s_{P1}, s_{P2}, \ldots, s_{Pm})$, then there is a sequence of transitions through L_i that corresponds to local sequence s_{Pi}.

For the if part, the question is whether the synchronizations in the synchronized steps of sequence Q must be feasible for M if each local sequence s_{Pi} of Q is feasible for L_i. Suppose *e* is one of the (possibly many) first infeasible events in sequence Q. There are three cases in which *e* can be infeasible:

Case 1: Event *e* is an asynchronous send event performed by ALTS *L*: Since *e* is a non-blocking send event, the only way for *e* to be infeasible is for ALTS *L* to be unable to perform *e*, but this contradicts the assumption that local sequence s_P is feasible for *L*.

Case 2: Event *e* is an asynchronous receive event executed by ALTS *L*, where *C* is the ALTS performing the asynchronous send *e'* synchronized with *e*. In order for *e* to be infeasible, at least one of the following is true:

(a) *L* is unable to execute receive event *e*, but this contradicts the assumption that s_P is feasible for *L*.

(b) *L* can perform receive event *e*, but the send partner *e'* for *e* cannot synchronize with *e*. But this contradicts the assumption that Q is feasible for M, as *e'* and *e* must be synchronizable (i.e., have matching labels) in order for Q to be feasible in M.

Case 3: Event e is a *synchronous-synchronization* event, where C is the ALTS performing the synchronous send e_s for e and U is the ALTS executing the synchronous receive e_r for e. In order for e to be infeasible, at least one of the following must be true:

(a) ALTS C is unable to perform e_s because ALTS C cannot execute a send event. But this contradicts the assumption that local sequence s_C is feasible for C.

(b) ALTS U is unable to perform e_r because ALTS U cannot perform a receive event. But this contradicts the assumption that local sequence $\mathrm{s_U}$ is feasible for U.

(c) ALTS C can perform send event e_s and ALTS U can perform receive event e_r, but e_s and e_r cannot synchronize with each other. However, this contradicts the assumption that Q is feasible for M, as e_s and e_r must be synchronizable (i.e., have matching ports and labels) in order for Q to be feasible in M. □

Theorem 4.4: CP $\leq_{\mathrm{F}}$ M iff for every $\mathrm{i} \in \{1..\mathrm{n}\}$ $P_i \leq_{\mathrm{F}} L_i$.

Proof: For the only-if part, Theorem 4.3 says that if a feasible sequence $Q = (s_{L1}, s_{L2}, ..., s_{Lm})$ of CP is feasible for M, then local sequences s_{Li} of Q are feasible for the individual ALTSs L_i of M, $1 \leq i \leq n$. It follows directly from this that if all the feasible sequences of CP are feasible for M, then all of the constrained local sequences of P_i are also feasible for the individual ALTSs L_i of M, $1 \leq i \leq n$.

For the if part, assume relation $P_i \leq_{\mathrm{F}} L_i$ holds but relation CP $\leq_{\mathrm{F}}$ M does not. Then there is an event e that is one of the "first" events in some feasible sequence $Q = (s_{L1}, s_{L2}, ..., s_{Lm})$ of CP that is not feasible for M. (An event e is one of the first infeasible events in Q if no event in Q that happened-before [Lamport 1978] e is infeasible. The first events are executed concurrently.) Assume that e is executed by thread P_j. Sequence s_{Pj} is a local sequence of P_j that is not feasible for L_i due to e, but we are assuming $P_j \leq_{\mathrm{F}} L_j$, which is a contradiction.

□

Theorem 4.5: M $\approx_{\mathrm{F}}$ CP iff for every $\mathrm{i} \in \{1..\mathrm{n}\}$ $L_i \approx_{\mathrm{F}} P_i$.

Proof: Theorem 4.5 can be rewritten as M $\leq_{\mathrm{F}}$ CP and CP $\leq_{\mathrm{F}}$ M iff for every $\mathrm{i} \in \{1..\mathrm{n}\}$ $L_i \leq_{\mathrm{F}} P_i$ and $P_i \leq_{\mathrm{F}} L_i$. From Theorem 4.2, M $\leq_{\mathrm{F}}$ CP iff for every $\mathrm{i} \in \{1..\mathrm{n}\}$ $L_i \leq_{\mathrm{F}} P_i$. From Theorem 4.4, CP $\leq_{\mathrm{F}}$ M iff for every $\mathrm{i} \in \{1..\mathrm{n}\}$ $P_i \leq_{\mathrm{F}} L_i$. This proves the Theorem.

A.2 Proof of Theorems 6.1, 7.1, and 8.2

Theorem 6.1: Let Q be a feasible SR-sequence of a model M that is comprised of a set of component ALTSs $\{L_1, L_2, \ldots, L_n\}$. Assume that every state in each component ALTS satisfies restrictions (R1) and (R2). Then each selection that a component ALTS makes is specified by Q and is deterministic.

Proof: SR-sequence $Q = (s_{L1}, s_{L2}, \ldots, s_{Ln})$, where s_{Li}, $0 < i \leq n$, is the totally-ordered sequence of send and receive events that occurred on ALTS L_i. Assume Q specifies that an event e is executed by some component ALTS L_i and assume that e is executed in state s of component L_i. The selection made by L_i depends on event e:

- Event e is a send-transition t of state s: Based on restriction (R1) (in Section 6.2), since t is a send-transition of state s, t is the only transition of s. Thus, the selection made by component L_i is deterministic.
- Event e is a receive-transition t_1 of state s that has annotation *t1.annotation*: Based on restriction (R1), all of the transitions of state s are receive-transitions. Based on restriction (R2), for any other receive transition t_2 of s, $t_1 \neq t_2$, *t1.annotation* $\neq$ *t2.annotation.* Thus, the selection made by component L_i is deterministic.

Theorem 7.1 deals with relation M $\approx_F$ CP, which requires that relations M $\leq_F$ CP and CP $\leq_F$ M both hold. Step (b1) of procedure $Test_{\approx F}$ in Fig. 11 verifies relation M $\leq_F$ CP. Recall that for each test sequence s of M, step (b1) shows that s is feasible for CP by using a controlled execution of CP with s. Assume the corresponding feasible sequence executed by CP is s'. Sequences s and s' execute the same events, have the same synchronization pairs of send and receive events, and have the same space time diagrams. This also means that the same happened-before relations exist between the events in sequence s and between the corresponding events in s'.

To prove Theorem 7.1, we must prove that relation CP $\leq_F$ M is also verified by procedure $Test_{\approx F}$. We first present two lemmas about the send events executed by CP. The first lemma considers whether non-determinism during the execution of s' can enable CP to execute a different event when it executes some send event in s'.

Lemma A.2.1: Assume that CP has executed all the events in *s'* that happen-before send event *snd*. When CP executes send event *snd* in *s'*, CP cannot exercise any other event.

Proof: In Section 5.3.1, we assumed that the only source of non-determinism during an execution is the order in which racing messages sent by two or more threads are received by the receiving thread. In particular, we assumed that the implementation language of CP does not allow a non-deterministic selection between two different send statements or between a send and a receive statement. Thus, when CP executed *snd* there was no other send or receive statement that CP could have executed. The value of the message sent at *snd* is deterministic and is completely determined by the events that happen-before *snd* in *s'*. Thus*, w*hen CP executes send event *snd* in *s'*, CP cannot exercise any other event.

When CP executes a receive event, the message received and port accessed partially depends on non-deterministic races between the send events that are available to be received.

Definition A.2.1: Let *Q* be a feasible sequence of model M or implementation CP and *rcv* be a receive event in *Q*. The set of available send events for *rcv*, denoted *rcv.available_sends*, is defined in the same way as the *race_set* of *rcv* in Proposition 1 of Section 5.1, except that condition (1) is dropped, i.e., send event *snd* is not required to be open at *rcv*.

Since set *available_sends* for receive event *rcv* in *Q* may contain send events that are not open, and hence are not in the *race set* of *rcv*, *rcv.race_set* is a subset of *rcv.available_sends*.

Lemma A.2.2: For each receive event *r* in feasible sequence *s* of M and the corresponding receive event *r'* in feasible sequence *s'* of CP, *r.available_sends* = *r'.available_sends*.

Proof: We show that if *r.available_sends* contains send event *snd* then *r'.available_sends* must contain the corresponding event *snd'*. The same argument can be used to show that if *r'.available_sends* contains send event *snd'* then *r.available_sends* must contain the corresponding event *snd*.

By way of contradiction, assume that *r.available_sends* contains send event *snd* but that the corresponding event *snd'* is not in *r'.available_sends*. Then *snd* satisfies conditions (2) – (4) in Proposition 1 in Section 5.1 for *r*, but *snd'* does not satisfy at least one of conditions (2) – (4) for *r'*. Thus, at least one of the following must be true:

(!2) *r’* happened-before *snd’* in sequence *s’* (while not *r* happened-before *snd* in sequence *s*)

(!3) *snd’* is synchronized with *d’* in sequence *s’* but not *r’* happened-before *d’* in *s’* (while *snd* is synchronized with *d* in sequence *s* and *r* happened-before *d* in *s*)

(!4) there is a send event *snd2’* that has the same source and destination as *snd’* and that happened-before *snd’*, but there does not exist a receive event *r2’* such that *snd2’* is synchronized with *r2’* and *r2’* happened-before *r’*. (On the other hand, there is a send event *snd2* that has the same source and destination as *snd* and that happened-before *snd*, but there exists a receive event *r2* such that *snd2* is synchronized with *r2* and *r2* happened-before *r*.)

Note that each of these conditions requires the events in sequences *s* and *s’* to be different, or the synchronization pairs in *s* and *s’* to be different, or the happened-before relations among the events in *s* to be different from the happened-before relations among the events in *s’*. However, since *s* is feasible for M and *s’* is feasible for CP, *s* and *s’* execute the same events; and *s* and *s’* have the same synchronization pairs and space time diagrams; and the same happened-before relations exist between the events in *s* and between the corresponding events in *s’*. This is a contradiction that proves the lemma.

□

Lemma A.2.2 says that when CP executed receive event *r’* in *s’*, the available sends for *r’* were the same as the available sends for the corresponding receive event *r* in *s*. Recall that an available send on port *p* can be received only if port *p* is open (i.e., if CP can execute a receive statement that accesses port *p*), and an available send for *r’* that is open is in the race set of *r’*. Therefore, if the send events that can be matched with *r’* are different from the send events that can be matched with *r*, this difference should be reflected in the race sets of *r’* and *r*. We use this to prove Theorem 7.1.

Theorem 7.1: If procedure $Test_{\approx F}$ is performed and all the tests in $\text{Feasible}_{\text{M}}$ receive a verdict of *pass* then $\text{M} \approx_{\text{F}} \text{CP}$.

Proof: Let *v’* be a variant (see Definition 5.8) of *s’* that is created by changing a receive event *r’*. Recall that v’ is a sequence of events that CP can execute instead of *s’*, given the inputs of *s’* and the available sends for the receive events in *s’*. From Section 2.2, the inputs of *s’* are the values specified in the input events of *s*’. Variant *v’* is said to be *invalid* if the corresponding sequence *v* is infeasible for M. Theorem 7.1 says that *v’* cannot be an invalid variant of *s’* if *r.race_set* = *r’.race_set*; otherwise *v’* represents a sequence of events that is feasible for CP but not for M, meaning that CP allows an extra behavior.

Assume by contradiction that variant *v'* is invalid and that *r.race_set* = *r'.race_set*. Then by the definition of variant in Definition 5.8, the send event *snd'* matched with *r'* in *v'* must be different from the send event matched with *r'* in *s'*. Furthermore, since *v'* is invalid, it must be impossible for the corresponding receive event *r* in sequence *s* to be matched with the corresponding send event *snd* in *s*. However, the fact that *snd'* can be matched with *r'*, but *snd* cannot be matched with *r*, means that *snd'* is in *r'.race_set* but *snd* is not in *r.race_set*. This contradicts the assumption that *r.race_set* = *r'.race_set*, which proves the theorem. □

Theorem 8.2: If procedure $LocalTest_{\approx F}$ is performed and all the tests in Constrained-Sequences(L$_{i}$) receive a verdict of *pass* then M $_{\approx F}$ CP.

Proof: Based on Theorem 7.1, race sets can be compared during global testing to verify relation M $_{\approx F}$ CP. For each receive event in a global test sequence and the corresponding receive event in the sequence exercised by the implementation, a pass/fail verdict is assigned based on the equality of the two race sets. Here we show that the verdicts assigned using global and local test sequences will be the same.

Let *s* be a global sequence *s* of M and s_{Li} be a local sequence that is projected from *s*. Assume receive event *r* appears in s_{Li}, which means that *r* also appears in *s*. Then the race set for *r* in sequence *s* is the same as the race set of *r* in s_{Li}. This is because the race set of *r* in sequence *s* is retained by *r* in s_{Li} when *s* is projected to create s_{Li}.

Now let *s'* be the global sequence exercised by CP that corresponds to sequence *s* of M, and s'_{Pi} be the local sequence exercised by P$_{i}$ that corresponds to local sequence s_{Li}. We have just shown that a receive event *r* that is in sequence *s* and s_{Li}, has the same race set in *s* and s_{Li}. Thus, if the verdicts assigned by global and local testing are different, it must be because the corresponding receive event *r'* executed by the implementation has different race sets in *s'* and s'_{Pi}. We show that this is not possible. Assume by way of contradiction that *r'* has different race sets in *s'* and s'_{Pi}. Then one or both of the following must be true:

(1) receive event *r'* has a different *OpenList* in *s'* and s'_{Pi}. However, the *OpenList* of *r'* depends only on the events executed by P_i that happen-before *r'*, and these events are the same in *s'* and s'_{Pi}.

(2) the happened-before relations among the send events sent to P_i are different in *s'* and s'_{Pi}. However, if *snd'* is a send event in *s'* that is sent to P_i, then the timestamps for event *snd'* in *s'* and for event *snd'* executed by the test driver are the same, since the timestamps for the *send* events in sequence *s* are saved and used by the test driver when s'_{Pi} is executed.

Thus, neither (1) nor (2) can be true, which is a contradiction that proves the theorem.