Department of Computer Science
George Mason University
Technical Report Series

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/
703-993-1530

A Modular Approach to Model-Based Testing of Concurrent Programs

Richard Carver
rcarver@cs.gmu.edu

Yu Lei
ylei@cse.uta.edu

Technical Report GMU-CS-TR-2013-5

Abstract

This paper presents a modular approach to testing concurrent programs that are modeled using labeled transition systems. Correctness is defined in terms of an implementation relation that is expected to hold between a model of the system and its implementation. The novelty of our approach is that the correctness of a concurrent software system is determined by testing the individual threads separately, without testing the system as a whole. We define a modular implementation relation on individual threads and show that modular relations can be tested separately in order to verify a (non-modular) implementation relation for the complete system. Empirical results indicate that our approach can significantly reduce the number of test sequences that are generated and executed during model-based testing.

1. INTRODUCTION

A concurrent program contains two or more threads that communicate and synchronize with each other to perform some task. One general approach to testing a concurrent program is to execute the program with carefully selected test sequences. *Model-based testing* uses abstract models for test selection. That is, an abstract model is used to specify the intended program behavior, and test-sequences selected from the model are used to test a concrete implementation.

Abstract 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 program events performed during state transitions. Our objective is to use model-based testing to determine whether a desired *implementation relation* exists between an abstract LTS model M and a concrete implementation CP. An example of such a relation is that the sequences of events allowed by M are also allowed by CP. When the implementation relation holds, we say that M is implemented by CP.

Black-box techniques have been developed for selecting test sequences from LTS models. However, concurrent threads execute non-deterministically during black-box testing. This makes it difficult to conclusively determine whether a selected test sequence is or is not allowed by the implementation. Gray-box testing techniques also generate test sequences from LTS models, but they require special testing tools that control inter-thread synchronization and force the implementation to deterministically execute selected test sequences [1-3]. Most black- and gray-box, model-based testing techniques use an interleaving concurrency model. This creates an explosion in the number of modeled states, which prevents a complete state space from being built, and an explosion in the number of modeled sequences, which makes it impractical to execute the implementation with all the test sequences.

In this paper, we present a modular approach to model-based testing for concurrent programs that use message passing for communication and synchronization. We assume the existence of an abstract model M containing two or more LTSs, and a concrete implementation CP with two or more concurrent threads. Our approach constructs a thread interaction model for each thread in the implementation. A thread interaction model is an annotated labeled transition system (ALTS) that captures the intended interactions between a single thread P and the other threads in M. A thread interaction model for P is typically much smaller than global model M, and can be constructed in an incremental manner. The thread interaction models are then traversed to derive abstract, modular test sequences for each thread. The annotations in the ALTSs are used to translate the abstract test sequences into concrete sequences that can be used to test P. The salient contributions of this paper are:

- A modular implementation relation: We define a new type of implementation relation, called a modular implementation relation, between a single implementation thread of implementation CP and the corresponding LTS(s) in model M. We show how to verify a global implementation relation between M and CP by verifying modular implementation relations between individual implementation threads and their LTS model(s).
- A modular testing technique: Modular implementation relations are verified by generating test sequences from M and testing each implementation thread in CP separately. The number of tests generated for an individual thread is typically small, so the total number of tests generated for modular testing may be significantly less than the number of tests generated by non-modular techniques. Also, since only one implementation thread executes during modular testing, modular testing does not require control over inter-thread synchronization and it never produces inconclusive test results.
- An implementation of modular testing: Modular-testing has been implemented using the Modern Multithreading class library [4]. The library contains thread and synchronization classes that provide testing and debugging services for multithreaded programs.

Our test generation technique makes one important assumption, which is that the sole source of non-deterministic behavior in the model and the implementation is the order in which LTSs/threads synchronize and communicate. Other sources of non-deterministic behavior, such as uninitialized variables in the implementation, are assumed to be absent. This assumption is discussed in detail in Section 2.

To illustrate our contributions, consider an LTS model and Java implementation of a well-known distributed mutual exclusion (DME) algorithm [5]. An interleaving model for DME contains more than 3.5 trillion sequences. A partial order model that we generated for DME contains 4,032 (non-modular) sequences. The stateless, modular testing technique presented in this paper produces only 315 modular test sequences for DME.

Model-based testing requires a model to be written at a suitable level of abstraction. One approach is for the model to express the behavior that a user is expected to observe. This requires a distinction to be made between observable and unobservable events in the implementation. The model is expressed using observable events only. Model-based testing is used to verify that the implementation and model have the same observable sequences of events. For example, for the DME implementation described above, a high-level model of *what* DME does can be written using $enter_i$ and $exit_i$ events for each of the three processes. This very abstract model admits only a handful of sequences. The DME implementation, however, can execute over 1.4 billion sequences – hundreds of millions of sequences for each abstract sequence generated from the model. It is not possible to obtain conclusive test results for the DME implementation with test sequences that abstract away all of the information about *how* the DME implementation achieves mutual exclusion.

Our approach is to develop an LTS model that includes the observable events of the implementation, i.e., events that represent interactions with the environment, and events internal to the implementation. These models can be verified against very abstract models, which are typically expressed using observable events only, and then used to test the implementation. This does not require the model and the implementation to be written at the same level of abstraction. The labels for the events in the LTS and the events executed by the implementation are simple strings that abstract the meaning and structure of program events. We add event annotations to the LTS model to supply the event details that are needed for test execution. The models can be written in specification languages that are problem oriented, not implementation oriented, and in specification styles [6] that express solutions using implementation-independent structures. These specification languages need not provide all the details that are provided by programming languages, the latter being concerned with things like efficient execution and opportune code reuse through inheritance.

The remainder of this paper is organized as follows. In Section 2, we show how the intended behavior of a concurrent program is modeled during modular testing. Section 3 defines a modular implementation relation and describes how modular testing can be used to verify this relation. Section 4 presents a technique for generating modular test sequences. Section 5 reports the results of an empirical study on modular testing. Related work is described in

Section 6, including a comparison between two other modular testing techniques and our technique. Section 7 provides concluding remarks and our plans for future work.

2. LTS MODELS

The modular testing technique presented in this paper is for concurrent programs that use message passing for communication and synchronization. The intended execution behavior of a message-passing program is modeled using an extended LTS model called an annotated LTS [7]. The LTS and annotated LTS models are described below.

2.1 Labeled Transition Systems

LTS models contain nodes representing the state of a program and labeled edges representing transitions from state to state. LTSs can be composed in parallel and they can be synchronized by performing *matching* send and receive events, where *e* represents a synchronous receive event that matches synchronous send event *e'*. Synchronizations involving matching events are considered to be hidden from external observers and are represented by a special τ (pronounced "tau") event. We assume that LTSs are composed using the laws in CCS (Calculus of Communicating Systems) [8] and an interleaving concurrency model.

Formally, an LTS is a 4-tuple $<Q,E,R,q_0>$, where Q is a non-empty finite set of states, E is a set of transition labels, $R \subseteq Q \times E \times Q$ is the transition relation, and q_0 is a state in Q denoting the initial state. For message-passing programs, the labels in E encode send and receive events. An LTS may contain one or more *termination states*, which are states without outgoing transitions. The τ events in an LTS are called *internal* events. All other events are called *external* events. A transition labeled with an internal (external) event is referred to as an internal (external) transition.

2.2 Adding Annotations

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, such as the ID of the sending or receiving thread, the operation performed, and the destination or source port of the operation [9], [2], [10]. (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.) Event descriptors are not included in LTS models, but they are needed to transform an abstract sequence of the model into a concrete sequence of the implementation, and to generate modular test sequences.

Koppol et al. [7] extended the LTS model and the algebraic laws used in CCS to allow implementation event descriptors to be encoded in an LTS. Their extended LTS model is called an annotated labeled transition system (ALTS). We use the ALTS model in this paper.

A formal definition of the ALTS model is given in [7]. Informally, an ALTS is an LTS in which each transition is annotated with information about the associated synchronization event in the implementation. A transition annotation has the form (L_i,L_j,port,op,label), where L_i is the sender and L_j the receiver for an operation *op* performed on *port* and labeled *label*. For synchronous message passing, *op* is either *synch_send* or *synch_receive*, or a synchronization between a *synch_send* and a *synch_receive*, denoted as a *synchronous-synchronization*. For asynchronous message passing, *op* is either *asynch_send* or *asynch_receive*. We use $E_{annotated}$ to denote the set of transition annotations in an ALTS. A technique for generating the annotations in an ALTS is described below.

The values for L_i and L_j in an annotation are handled differently for synchronous and asynchronous message passing. For synchronous message passing, the sending and receiving LTSs are modeled as having a direct synchronous interaction with each other, so the values for L_i and L_j refer to the IDs of the LTSs modeling the sender and receiver threads, respectively. Since LTS models use synchronous message passing semantics, asynchronous message passing must be simulated using synchronous communication. This is done by having the sending and receiving LTSs interact with an LTS that models a communication medium between the sender and receiver. Note, however, that the values of L_i and L_j refer to the IDs of the LTSs modeling the sender and receiver threads, respectively, not the medium involved in the communication. Medium objects are modeling artifacts that do not exist in the implementation.
We assume that each send or receive event in the implementation is expected to appear in the ALTS model of the implementation. The ALTS 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 also assume that these extra events are removed from test sequences generated from the model before the sequences are used to test the implementation.

ALTs are composed using the laws in CCS, with extra rules about forming annotations for synchronizations. For example, consider a composition of ALTS B_1 and ALTS B_2, denoted by *(B_1 | B_2) \ {msg_in}*. A synchronization between B_1 and B_2 on receive event *msg_in (B1,B2,in,synch_receive,msg_in)* and send event *msg_in'(B1,B2,in,synch_send,msg_in)* results in a τ event with annotation *(B_1,B_2,in,synchronous-synchronization,msg_in)*. The annotation for the τ event denotes that B_1 was the sender and B_2 the receiver for a synchronous synchronization labeled *msg_in* that occurred on port *in*. The annotation for the τ event carries the annotation information from the events that were synchronized to create it.

The sole source of non-deterministic behavior in an LTS of the model, or a thread in the implementation, is assumed to be the non-deterministic order in which sent messages can be received. Other possible sources of non-determinism in the implementation, such as uninitialized variables, or accessing memory after it has been de-allocated, are assumed to be absent. Non-deterministic selections between an input statement and an output statement are not allowed. Likewise, non-deterministic selections between two *send* statements, or between two receive statements that access the *same port*, is not allowed. These latter types of non-deterministic selections could potentially be modeled as a state in an ALTS that has two or more outgoing transitions that have the same event label and annotation. However, these types of selections are typically not allowed by implementation-level message-passing constructs, and we do not find such selections useful at the implementation level.

We point out that some types of non-determinism that are not useful at the implementation level 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 [11], [12]. We do not discourage this type of design-level non-determinism. It can be used in earlier stages of modeling as long as it is reduced before the model is used to generate test sequences, reflecting the fact that all of the design decisions have been made.

The events in an ALTS model M may represent message-passing between threads, or they may instead represent I/O operations (e.g., reading from a keyboard, or file) between the threads and their environment. Model M thus identifies the inputs that are used to verify the implementation relation between M and 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 *s*. We assume that when sequence *s* is used to test CP, the inputs of *s* are translated into the required input format for CP.

Annotations must be generated in order to create an ALTS model instead of an LTS model. Chen and Carver [13] showed how annotations can be generated for models written in the Lotos specification language [14]. Annotations are automatically computed when the Lotos model is compiled into an LTS. The annotations appear as part of the LTS transition labels. 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 5 to create ALTS models from Lotos models. Refer to Chen and Carver [13] for details about this approach.

3. MODULAR TESTING

Modular test generation begins with an abstract model M comprised of a set of ALTSs {L_1, L_2, …, L_m}, and a concrete implementation CP of M with concurrent threads {P_1, P_2, …, 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 an unreliable communication medium for which there is no equivalent thread in CP. Ultimately, we require each of the *n* threads in CP to be mapped to the ALTS(s) in M that the thread implements. 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 that the alphabets of event labels for P_i and L_i are intended to be the same. Our objective is to determine whether a specified implementation relation exists between M and CP by generating test sequences from M and using these sequences to test the threads in CP.

3.1 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 = <Q,E,R,q_0> of CP. The set of all possible sequences that can be written using the labels in set $E_{annotated}$ of model M is denoted by $E_{annotated}$*.

Definition 1: A sequence *s* in $E_{annotated}$* is *feasible for model M* if *s* is a sequence of events along some path through M, starting at the start state of M; otherwise, *s* is *infeasible* for M.

Definition 2: A sequence *s* in $E_{annotated}$* is *feasible for implementation CP* if an execution of CP can exercise sequence *s*.

A *non-modular* implementation relation that is often used for test generation is denoted by $M \le_F CP$.

Definition 3: $M \le_F CP \equiv_{def}$ for any sequence *s* in $E_{annotated}$*: *s* is feasible for M $\Rightarrow$ *s* is feasible for CP.

Relation $M \le_F CP$ 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 perhaps that M is incomplete and thus is *extended* by CP, i.e., CP adds behavior that is not in M, but all the behaviors of M are still allowed by CP [11].

3.2 A Modular Implementation Relation for $\le_F$

The implementation relation in Definition 3 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 $L_i = <Q_i,E_i,R_i,q_i>$ in M to which P_i is mapped.

Definition 4: A *local* sequence with respect to ALTS L_i is a sequence $s_{Li} \in E_{annotated}$* such that all of the send and receive events in s_{Li} have L_i as the sender or receiver, respectively.

A sequence that is local with respect to ALTS L_i may or may not be allowed by L_i.

Definition 5: Let s_{Li} be a sequence that is local with respect to L_i. Local sequence s_{Li} is *feasible for L_i* if s_{Li} is a sequence of events along some path through L_i, starting at the start state of L_i; otherwise, s_{Li} is *infeasible* for L_i.

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 can be sent to L_i.

Definition 6: For feasible sequence *s* of M, the *projection of s onto L_i* is the (feasible) local sequence s_{Li} that is obtained by removing from *s* all of the send events for which L_i is not the sender and all of the receive events for which L_i is not the receiver.

- If *e* is an *asynch_send (asynch_receive)* event in *s* that is executed by L_i, then *e* is an *asynch_send* (*asynch_receive*) event in s_{Li}.
- If *e* is a *synchronous-synchronization* event in *s*, then *e* is a *synch_send* (*synch_receive*) event in s_{Li} if L_i executed the *synch_send* (*synch_receive*) event synchronized at *e*.

Definition 7: A feasible local sequence s_{Li} of L_i is *constrained with respect to model M* if s_{Li} is the projection onto L_i of some feasible sequence of M. 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.

Definition 7 shows that Constrained-Sequences(L_i) is not determined by analyzing L_i and ignoring the other ALTSs in M. To the contrary, a sequence in Constrained-Sequences(L_i) must be a projection of some feasible sequence of M. Thus, Constrained-Sequences(L_i) captures the constraints imposed on L_i by the other ALTSs.

When a feasible sequence s of M is projected to obtain a constrained local sequence s_{Li} of L_i, the annotations on the events in *s* are retained by the events in s_{Li}. These annotations 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 exercises a matching send (receive) event. An environment that interacts as specified by the annotations in s_{Li} is referred to as a *conforming* environment of s_{Li}.

Definition 8: A feasible 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 3.3.

Theorem 1: Let *s* be a feasible sequence of M and s_{Li} be the projection of *s* onto ALTS L_i, $1 \le i \le n$. Then constrained local sequence s_{Li} is feasible for thread P_i, $1 \le i \le n$, iff sequence *s* is feasible for CP.

Proof: See Appendix 1.

Based on Theorem 1, we can test each thread separately with the *constrained* local sequences of its corresponding ALTS instead of testing all the threads together with all of the feasible sequences of M. Building on this, we define a modular implementation relation for an ALTS L_i and the thread P_i to which it is mapped.. This relation mirrors the relation in Definition 3:

Definition 9: $L_i \le_F P_i \equiv_{def}$ for any sequence s_{Li} in Constrained-Sequences(L_i): s_{Li} is feasible for P_i.

Modular implementation relation $L_i \le_F P_i$ is used in the following theorem, which is the basis for modular testing:

Theorem 2: $L_i \le_F P_i$, $1 \le i \le n$, iff $M \le_F CP$.

Proof: The if-part is obvious – Theorem 1 says that if a feasible sequence *s* of M is feasible for CP, then the constrained local sequences obtained by projecting *s* onto L_i, $1 \le i \le n$, are feasible for the individual threads of CP. 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, $1 \le i \le n$, are also feasible for the individual threads of CP.

For the only-if part, assume relation $L_i \le_F P_i$, $1 \le i \le n$, holds but relation $M \le_F CP$ 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 *s* of M that is not feasible for CP. (An event *e* is one of the first infeasible events in *s* if no event in *s* that happened before [15] *e* is infeasible. The possible first events are executed concurrently.) Assume that *e* is executed by ALTS L_j and let s_{Lj} be the projection of *s* onto 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 $L_j \le_F P_j$, which is a contradiction.

According to Theorem 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 sequences of M.

We point out that our approach is modular 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 a reduced version of M, not just L_i. Thus, our approach is not modular 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 allow local sequences that are not projections of any of the feasible sequences of M. 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 $M \le_F CP$ is violated. Likewise, if these local sequences are infeasible for P_i, but they cause runtime assertions in P_i to fail during test execution, it does not imply that P_i has faults.

3.3 A Modular Testing Procedure for $M \le_F CP$

Modular testing is performed using the following procedure:

Procedure Test$_{\le F}$: For each mapped pair (L_i, P_i), $1 \le i \le n$:

(a) Generate Constrained-Sequences(L_i) (see Section 4.2).

(b) For each local test sequence s_{Li} in Constrained-Sequences(L_i):

- (b1) Test P_i with sequence s_{Li} and assign a test verdict, which is either *pass* or *fail*. The assignment of verdicts is discussed below.
- (b2) If P_i *fails* with s_{Li}, a failure has been detected in CP and testing halts.

Note that the key step in the above procedure is deriving Constrained-Sequences(L_i) in step (a), which is described in Section 4. In step (b1), thread P_i is executed with a test driver. The driver behaves as a conforming environment 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} calls for P_i to execute a send (receive) event on port *p*, a receive (send) event on port *p* is executed by the driver. The implementation information that is needed for mapping the abstract events in s_{Li} to concrete events of P_i is provided by the transition annotations in L_i, as described in Section 2. Note that the execution of thread P_i interacting with a test driver will be deterministic.

The test driver must supply message objects for the send events that it executes. To assist in this process, we can perform reachability testing on implementation CP and capture the message objects that are sent. During reachability testing, message objects are stored in a map structure that maps a message label to its associated message object. When the test driver needs to send a message with a given label, it uses 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 complete; reachability testing 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 test verdict in (b1) is assigned as follows:

if (P_i executes an event that is not in the alphabet E_i of L_i or local sequence s_{Li} is infeasible for P_i) then the test *fails* else the test *passes*.

If procedure Test$_{\le F}$ is performed and all the tests in Constrained-Sequences(L_i) are passed for each mapped pair (L_i, P_i), $1 \le i \le n$, then $L_i \le_F P_i$, and by Theorem 1, $M \le_F CP$. Note that when thread P_i is tested with the local sequences in Constrained-Sequences(L_i), the tests are used to determine whether P_i will interact as intended with the other threads in the program. There is no circular reasoning used in this approach — we do not assume that the other threads are correct when testing P_i (or that P_i is correct when each of 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 sequences of M. This is, however, not necessarily the case. For example, if ALTS L models a thread that interacts with all the other threads in the system, then each feasible sequence of M might correspond to a different local sequence of L and no reduction will be achieved by generating Constrained-Sequences(L). Such a result is reported in the case study in Section 5.

4. Modular Test Generation Using Thread Interaction Models

The objective of modular test generation is to generate a set of constrained local test sequences, Constrained-Sequences(L_i), for each ALTS L_i in model M. These test sequences are used in step (b) of procedure $\text{Test}_{\leq F}$ in Section 3.3. In this section, we show how to build an ALTS model called a thread interaction model (TIM). The thread interaction model for Li, denoted TIM_{Li}, models L_i's interactions with the other ALTSs in M. Constrained-Sequences(L_i) is generated by traversing TIM_{Li}.

One approach to generating TIM_{Li} is to use reachability analysis to build a global ALTS M_g for model M and then use an equivalence-based reduction to derive a TIM_{Li} that is smaller but equivalent with regard to L_i's behavior in global ALTS M_g. A second approach is to use incremental reachability analysis to build TIM_{Li} [16-17]. Incremental techniques also use equivalence-based reductions, but they do so without first generating M_g. As a result, incremental techniques may be much more efficient. The TIM_{Li} produced by non-incremental and incremental approaches is the same.

A number of equivalence relations have been defined for LTSs [8]. Observational equivalence is used to relate two LTSs whose behaviors are indistinguishable when their τ events are invisible. The process that we use in Section 4.1 to generate thread interaction model TIM_{Li} includes a reduction based on observational equivalence. LTS L_i's behavior in TIM_{Li} is indistinguishable from its behavior in M when events in M that do not directly involve L_i are unobservable. Another equivalence relation that we use is called weak-trace equivalence. Informally, two LTSs are weak-trace equivalent if they can perform the same sequences of external events, starting from their initial states. In Section 4.1, we use a reduction based on weak-trace equivalence to remove τ events and redundant transition sequences from thread interaction models before we generate test sequences from the models. This ensures that no duplicate test sequences are generated and no τ events appear in the test sequences. Formal definitions of the observational and weak-trace equivalence relations, and reduction algorithms for these relations, can be found in [18] and references therein.

4.1 Using Reachability Analysis to Generate Thread Interaction Models

For each ALTS L_i in model M, we use equivalence-based reductions to build a thread interaction model TIM_{Li} that models L_i's interactions with the other ALTSs in M. The steps for building TIM_{Li} are as follows:

Step 1: Based on ALTS L_i, classify the transitions in model M as observable or hidden.

For asynchronous message passing, the observable transitions are the send and receive transitions executed by L_i. For synchronous message passing, the observable transitions are the transitions that involve a synchronization in which L_i is the sender or the receiver. Other transitions are considered to be hidden. Thus, the observable transitions in M all involve interactions with L_i.

Step 2: Minimize M modulo observational equivalence creating ALTS model M_{Li}, which captures L_i's behavior in M.

Step 3: Minimize M_{Li} modulo weak-trace equivalence creating reduced ALTS TIM_{Li}.

When a minimization is performed in Step 2 or 3 the minimization is based on the *annotations* in the ALTSs. Recall that synchronizations between L_i and the other threads in M are labeled as τ events. If minimization were to be based on transition labels, instead of annotations, τ events could be removed during minimization, which would allow information about the original synchronizations to be lost. Since minimization is based instead on annotations, all the τ events will have the same label "τ" but different annotations. This allows τ events to be treated as different (observable) events during minimization. Thus, the annotation information about the events and the ALTSs that synchronize with L_i is retained in TIM_{Li}. This ensures that TIM_{Li} models all of L_i's interactions with other threads, and that TIM_{Li} contains the implementation information (in the form of annotations) that is necessary for generating concrete test sequences for implementation thread P_i.

The reduced thread interaction model TIM_{Li} produced in Step 3 represents the feasible sequences of interactions between L_i and the rest of the system. TIM_{Li} may be non-deterministic due to concurrent interactions between L_i and the other components. Based on the definition of weak-trace equivalence, TIM_{Li} contains no hidden transitions and no redundant sequences of observable transitions. Algorithms for Step 2 run in time $O(n^3)$, where n is the number of states in the model. Algorithms for Step 3 have worst case running times that are exponential in the number of states, but the model minimized in Step 3 is a model of a single thread, which is typically much smaller than a global model. Thus, Step 2 should dominate the execution time.

4.2 Generating Test Sequences from Thread Interaction Models

Test sequences are derived by traversing TIM_{Li} and generating all the feasible sequences. For a cyclic model such as $TIM_{Li} = a.TIM_{Li}$, an exhaustive test suite would have infinitely many test sequences, each test sequence *a*, *a.a*, *a.a.a*, …, having a finite but arbitrarily long number of events. A similar type of problem occurs when generating (white-box) tests from implementations that have loops (while-loops, for-loops, etc) [19]. (Note that an implementation that has a loop does not necessarily have a cycle in its state space.) One approach for dealing with a cyclic model M is to select a finite subset of M's test sequences, and ensure that cycles are iterated a finite number of times. Another approach is redesign M as a model M' that is incomplete but that has an acyclic state space, and generate an exhaustive test suite from M'. In this case, the feasible sequences of acyclic model M' form a subset of the feasible sequences of the cyclic model M. In both approaches, a finite set of (finite-length) test sequences is generated. However, the test sequences may fail to detect some errors.

If the TIM_{Li} generated by the process in Section 4.1 is *acyclic*, then it can be traversed using a simple depth-first search (DFS) algorithm. Whenever a termination state of TIM_{Li} is reached, the DFS backs up, and one or more test sequences are collected from the search stack. The collected test sequences include one complete sequence of TIM_{Li}, i.e., a sequence beginning at the start state of TIM_{Li} and ending in a termination state, and all of the non-null proper prefixes of this complete sequence. For example, if the search stack contains (from the bottom of the stack to the top) *a b c* when the search backs up, then the complete sequence *a.b.c* is generated along with prefix sequences *a.b* and *a*. Thus, the DFS will generate all the possible traces of TIM_{Li}.

If TIM_{Li} is cyclic, then some test selection method must be used when TIM_{Li} is traversed to select a subset of the feasible test sequences of TIM_{Li}. Test selection may be based on, e.g., guidance from the person doing the testing [20-22], or coverage criteria [19].

Modular sequences are generated for each ALTS L_i in M using the following procedure:

```
Procedure Generate_Sequences(L_i):
  For each ALTS L_i in model M:
  (G1) Apply Steps 1 through 3 in Section 4.1 to
  create thread interaction model TIM_Li
  (G2) Traverse TIM_Li as described above to
  generate test sequences for L_i.
```

Whether procedure *Generate_Sequences(L_i)* generates Constrained-Sequences(L_i) depends on whether TIM_{Li} is acyclic.

Theorem 3: If TIM_{Li} is *acyclic*, then procedure *Generate_Sequences(L_i)* generates Constrained-Sequences(L_i) for each ALTS L_i, $1 \le i \le n$, in model M.

Proof: In procedure *Generate_Sequences(L_i)*, minimization modulo weak-trace equivalence in step (G1) is based on the annotations of the external transitions instead of the transition labels. This prevents any (non-redundant) sequences of L_i's transitions from being lost during the minimization. By definition of weak-trace equivalence, the traces of TIM_{Li} generated by the DFS procedure in step (G2) are precisely Constrained-Sequences(L_i).

We point out that an obvious optimization of the DFS search procedure is to avoid the generation of any sequence that is a prefix of a sequence that has already been generated. For example, if 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. This optimization is easily performed during DFS by generating only complete sequences at backup points.

If TIM_{Li} is cyclic, then procedure *Generate_Sequences(L_i)* must use some test selection method to select a subset of the feasible sequences of TIM_{Li}. Thus, *Generate_Sequences(L_i)* will not generate Constrained-Sequences(L_i) or even all of the complete sequences of L_i. In this case, modular testing cannot be used to verify relation $M \le_F CP$; however, the generated sequences will be *sound*, i.e., only incorrect implementations will fail the tests [23].

5. EMPIRICAL STUDY

We conducted an empirical study in which modular test sequences were generated from thread interaction models. Abstract models and Java implementations were built for:

DP: a deadlock-free solution to the dining philosophers problem with philosophers sharing forks and each philosopher eating once [4]. In this solution, all philosophers but one pick up their left fork first, while the one “odd philosopher” picks up its right fork first.

DME-3: a solution to the distributed mutual exclusion problem with three processes and three threads per process [5]. Processes communicate using asynchronous message passing. 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 resource one time.

TDME: a token-based solution to the distributed mutual exclusion problem [24] with 3 user processes and 1 controller process. A Process that wishes to have exclusive access to a shared resource must obtain a special token from the controller process. Each process uses the shared resource one time.

All the models and implementations were acyclic. Note that while DME and TDME solve the same problem, they have significantly different synchronization behavior. Thread interaction models were generated by following the step-by-step procedure in Section 4.1 using the CADP toolset [25-26] and the ALTS reduction tool in [7]. We developed our own programs for generating and counting modular and non-modular test sequences from the ALTS models.

Our objective here is to study the effectiveness of modular tests for detecting violations of the implementation relations and to compare the number of test sequences generated by modular testing to the number of sequences generated by other approaches. The results show a range of results for reducing test set sizes, from a large reduction to no reduction. In this study, we leveraged several LTS reduction tools, and inherited their limitations, but we did not evaluate their scalability. That has been done by others [18], in many cases on real life, industrial systems.

Table I (see last page) summarizes the results of test sequence generation. For each of the three models, Table I shows:

(1) The number of states and transitions in the global ALTS (column 1). 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 non-modular test sequences (columns 2 and 3). Non-modular test sequences were generated using two different methods, which are described below.
(3) The number of states and transitions in the largest thread interaction model (TIM) (column 4).
(4) The total number of modular test sequences generated from the thread interaction models (column 5).

Each Lotos specification model was compiled into its individual ALTS components. The longest time for this step occurred while translating the Lotos DME-3 model into its 9 component ALTSs, which took a total of 7 minutes and 40 seconds on a 1.3GHx processor with 32 GB of RAM.

Non-modular test sequences were generated using two different methods. The first method reports the number of unique, *totally-ordered*, non-modular sequences generated from the ALTS models (column 2 of Table I). This is the number of sequences generated when concurrent events are modeled by enumerating their possible interleavings. We counted the totally-ordered sequences using the DFS procedure described in Section 4.1. For model DME-3, this procedure was unable to finish. Thus, we report our partial results as lower bounds on the number of sequences.

The second method reports the number of non-modular sequences generated by the reachability testing algorithm in [10] when it was applied to the DP, DME-3 and TDME models. This algorithm uses a true-concurrency model to generate the unique *partially-ordered* feasible sequences of the models (column 3 of Table I). The number of partially-ordered sequences is usually considerably smaller than the number of totally-ordered sequences, and should also be smaller than or competitive with the results achieved by partial order reduction [27-28] and true-concurrency [29-31] techniques. The non-modular, partially-ordered sequences can be used to verify that relation $M \leq_F CP$ holds.

Modular test sequences were generated from thread interaction models using the optimized DFS procedure described in Section 4.2. Since every transition in a thread interaction model involves the thread under test, no two transitions in the same modular test sequence are concurrent. This guarantees that no two modular sequences differ only in the order of concurrent events, the same as for partially-ordered sequences. Generating modular tests from the thread interaction models and executing the tests against the implementations took only a few seconds.

Table I shows that the number of modular test sequences was always significantly less than the number of totally-ordered sequences generated from the global models; and was significantly less than the number of partially-ordered sequences generated from three out of six of the global models. For TDME, DP, and DME-3, this is reflected in a comparison of the sizes of the global models and the thread interaction models, the latter being significantly smaller. The time for generating thread interaction models took 24 to 30 seconds.

The number of modular test sequences generated for model DP-P 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 considerably more totally-ordered sequences. When P = 6, there are 18 modular test sequences, 62 partially-ordered non-modular sequences and over 200 billion totally-ordered, non-modular sequences.

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

For the DME-3 model, a total of 315 modular test sequences were generated for the nine implementation units. The global DME-3 model has 4,032 partially-ordered non-modular sequences and over 3.5 trillion totally-ordered, non-modular sequences. We measured the adequacy of the test-sequences generated for DME-3 by using mutation testing. Each mutant for DME-3 introduced a single change that was intended to simulate a programming error. For DME-3, we generated a batch of mutants, which were the mutants identified by the Java-based mutation tool μJava [32]. Some of the mutants created were functionally equivalent to the original program. These mutants were identified and deleted, which left 190 mutants. We then applied modular testing to the nine threads in DME-3. A mutant was considered to be *killed* if a modular test sequence failed when it was executed against the thread that contained the mutant. Each of the 190 DME mutants was killed by the modular tests.

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 three programs are small in terms of lines of code, but they represent complex, classical synchronization patterns and they illustrate well that the reduction in test sequences achieved by modular testing will vary from none at all to a significant amount. We plan to conduct experiments on more programs as an effort to reduce this threat. The main threat to internal validity is the possibility that errors were made in counting the test sequences. The partially-ordered sequences were counted using the reachability testing tool in [10]. The totally-ordered sequences were counted using a depth-first search algorithm, whose implementation was carefully tested.

6. RELATED WORK

In this section, we briefly review existing work on model-based testing of concurrent systems. We note first that test sequences can also be generated by analyzing an implementation's structure [33-34] or its runtime behavior [27-28][35]. Model-based and implementation-based testing are complementary approaches — certain faults may be detectable when using an implementation-based approach but not when using a model-based approach, and vice versa.

We also note that there has been work in the area of compositional model checking. The basic idea is to verify the behavior of each module in isolation and infer global correctness properties of the whole system from the results of verifying individual modules. This typically requires the user to manually provide an assumption about the interaction between the module being checked and the rest of the system. Our modular testing technique automatically builds a thread interaction model using equivalence-based reductions of the whole system. Also, our test generation technique does not perform any model-checking. Model checking is complementary to our work — the abstract model can be verified using model checking before the model is used to generate modular tests for testing the implementation.

Most existing model-based testing techniques for concurrent systems use a finite state machine (FSM) model, such as an I/O automaton [36] or an I/O state machine [37], or they use an LTS model. These techniques have been developed mainly in the area of testing protocol implementations. Typically, the correctness of the implementation under test is defined in terms of a conformance relation between the implementation and its model. Conformance tests try to detect differences that are not allowed by the conformance relation.

In general, conformance testing techniques are *black-box* techniques. Black-box techniques are effective for testing a single thread, but they encounter problems when they are applied to a set of concurrent threads. One problem is that a set of non-deterministic threads executes non-deterministically, which produces inconclusive test results. Another problem is that the composite FSM and LTS models are often based on an interleaving model of concurrency, which creates the well-known state explosion problem in the size of the models, making test generation impractical. An alternative approach is to generate test sequences on-the-fly [23], so that the only part of the state space that is expanded is the part needed for the next test step. This approach, however, may still be impractical as the time needed to execute the implementation with the potentially huge number of interleaving sequences generated from the model may be prohibitive.

Along a different line, several techniques [31-32] have been developed for generating non-modular test sequences from true-concurrency models. In such models, a non-modular test sequence is a partial order of transitions in which concurrent transitions are left unordered. In contrast, our modular testing technique generates sequences for individual threads. As the case study shows, the total number of modular sequences may be significantly smaller than the total number of non-modular sequences, even when an interleaving-free approach is used to generate non-modular sequences.

Several compositional conformance testing techniques have been developed. Van der Bijl et al. [38] showed how to perform compositional conformance testing based on the *ioco* conformance relation can be performed. They determined a sufficient condition under which the *ioco* conformance of the component implementations to their respective LTS specifications leads automatically, without any additional testing, to the *ioco* conformance of the system implementation to the system specification. Roughly speaking, the sufficient condition is that each component's LTS model is *input enabled*. An LTS is input enabled if each state specifies a response for every possible modeled input. It is also assumed that for each implementation component all inputs are enabled in all states.

Gotzhein and Khendek [39] presented a compositional technique for testing protocol implementations that can be modeled as the concurrent composition of *two* input enabled, deterministic FSMs. Each of the two implementation components is tested separately using one of the traditional black box methods mentioned above. When the two components pass their local tests, test sequences are generated to test for 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 modular testing technique presented in this paper can be applied to models with more than two implementation components, and if the local tests are passed no separate integration tests are required.

An important issue with modular testing is whether the model of an individual thread specifies the exact set of inputs that can be received in a particular state. The modular techniques in [38] and [39], as described above, require each individual thread model to be input enabled, so that each state specifies a response for every possible modeled input. One problem with this approach is that some inputs may be impossible in certain states, and it is not clear what response should be specified for an impossible input. Also, identifying impossible inputs manually is difficult when the possible inputs depend on complex interactions among multiple components. Another problem 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. 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 modular 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 may contain states that allow inputs that are impossible in a global context, or that disallow some inputs that are available. Impossible inputs are not a problem in our framework, since the process used to create thread interaction models implicitly identifies impossible inputs and prevents them from being included in the modular test sequences that are generated.

A work closely related to ours is [7], which presents a 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 new ALTS reduction algorithm. The new reduction algorithm uses the transition annotations in the reduced ALTS to store information about the paths in the unreduced state space of M. When annotations are considered, 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 non-modular deterministic testing. Also presented in [7] are several modular coverage criteria — test sequences generated from reduced ALTSs can guarantee a level of coverage for the global model without ever having to build the global model. Satisfying these modular coverage criteria, however, does not ensure the satisfaction of the implementation relations defined in Section 3 of this paper.

7. CONCLUSION

In this paper, we presented a modular approach to testing concurrent systems that are modeled as annotated labeled transition systems. The novelty of our modular approach is that the correctness of a concurrent system is determined by testing the individual implementation threads separately, without testing the implementation as a whole. Correctness is defined in terms of a modular implementation relation that is expected to hold between the individual threads in an implementation CP and their corresponding ALTS(s) in model M. We defined a modular implementation relation and showed how to verify this relation using modular testing. If this modular relation is verified, then this also verifies that the (non-modular) sequences allowed by the complete model M are allowed by CP. Empirical studies confirmed that modular testing may require significantly fewer test sequences than non-modular testing.

We plan to continue our work by developing a modular testing technique that can be used to verify relation $M \leq_F CP$ and also relation $CP \leq_F M$. The objective is to use modular testing to check whether M and CP allow the same sequences.

REFERENCES

1. Carver, R., Tai, K.C.,: Replay and testing for concurrent programs. IEEE Software, 66-74 (1991)
2. Tai, K.C., Carver, R.H.: Testing of distributed programs. Chapter 33 of *Handbook of Parallel and Dist. Computing*, ed. by A. Zoyama, McGraw-Hill, 955-978 (1996)
3. Tai, K.C., Carver, R.H., Obaid, E.: Debugging concurrent Ada programs by deterministic execution. IEEE Trans. Software Engineering, 17(1):45-63 (1991)
4. Carver, R., Tai, K.C.: Modern Multithreading: Wiley. http://www.cs.gmu.edu/~rcarver/ ModernMultithreading/ (2006)
5. Ricart, G., Agrawala, A.K.: An optimal algorithm for mutual exclusion in computer networks. Comm. of the ACM, 24, 1 (January), 9-17 (1981)
6. Vissers, C, Scollo, G., Van Sinderen, M.: Architecture and specification style in formal descriptions of distributed systems. (Invited) In: Proceedings Eighth International Symposium on Protocol Specification, Testing, and Verification, 189-204 (1988)
7. Koppol, P.V., Carver, R.H., Tai, K.C.: Incremental Integration Testing of concurrent Programs. IEEE Transactions on Software Engi. Vol. 28, No. 6 (2002)
8. Milner, R., *Communication and Concurrency*, Prentice-Hall (1989)
9. Tai, K.C.: On testing concurrent programs. Proc. COMPSAC 85, 310-317 (1985)
10. Lei, Y., Carver, R.H.: Reachability testing of concurrent programs. IEEE Transactions on Software Engineering, Volume 32, No. 6, 382-403 (2006)
11. Brinksma, E.: A Theory for the Derivation of Tests. in: S. Aggarwal, K. Sabnani, eds., Protocol Specification, Testing and Verification, VIII, 63-74. (1988)
12. Chung, I.S., Kim, B.M., Kim, H.S.: A new approach to deterministic execution testing for concurrent programs. IEICE Trans. Inf. Syst. Vol. E84-D, No.12, 1756-1766 (2001)
13. Chen, J., Carver, R.: Selecting and Mapping Test Sequences from Formal Specifications of Concurrent Programs. Proc. of the High-Assurance Systems Eng. Workshop, 112-119 (1996)
14. Turner, K.J.: Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL. John Wiley & Sons, Inc., New York, N.Y (1993)
15. Lamport, L.: Time, Clocks, and the Ordering of Events in a Dist. System. Comm. ACM, 558-565 (1978)
16. Cheung, S.C., Kramer, J.: Enhancing Compositional Reachability Analysis with Context Constraints. Proc. 1st ACM SIGSOFT Symp. on Foundations of Software Eng., 115-125 (1993)
17. Cheung, S.C., Kramer, J.: Compositional Reachability Analysis of Finite-State Distributed Systems with User-Specified Constraints. Proc. 3nd ACM SIGSOFT. Symp. on Foundations of Software Eng.,141-150 (1995)
18. Cleaveland, R., Parrow, J., Steffen, B.: The Concurrency Workbench: A Semantics Tool for the Verification of Concurrent Systems. ACM Tran. Programming Languages and Systems, Vol. 15, no. 1, 36-72 (1993)
19. Ammann, P., Offutt, J.: *Introduction to Software Testing*. Cambridge University Press (2008)
20. Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework. International Standard IS-9646. ISO, Geneve (1991)
21. Feijs, L.M.G., Goga, N., Mauw, S., Tretmans, J.: Test Selection, Trace Distance and Heuristics. Proc. IFIP 14th Int. Conference on Testing Communicating Systems - TestCom, 267-282 (2002)
22. Tretmans, J., Brinksma, E.: TorX: Automated Model-Based Testing. Proc. First European Conference on Model-Driven Software Engineering, 31-43 (2003)
23. Tretmans, J.: Testing Concurrent Systems: A formal approach. Lecture Notes in Computer Science; Vol. 1664, Proc. of the 10th International Conference on Concurrency Theory, 46 – 65 (1999)
24. Suzuki, I., Kasami, T.: A distributed mutual exclusion algorithm. ACM Transactions on Computer Systems, 3(4): 344-349 (1985)
25. Fernandez, J., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., Sighireanu, M.: CADP: A Protocol Validation and Verification Toolbox. Proc. 8th Conf. on Computer-Aided Verification, 437-440 (1996)
26. Lang, F.: Compositional Verification using SVL Scripts. LNCS, Vol. 2280/2002, 127-136 (2002)
27. Godefroid, P.: Model Checking for Programming Languages using VeriSoft. Proc. 24th ACM Symp. on Principles of Prog. Languages, 174-186 (1997)
28. Flanagan C., Godefroid, P.: Dynamic partial order reduction for model checking software, Proc. 32nd Symposium on Principles of Programming Languages (POPL) (2005)
29. Jard, C.: Principles of Distributed Test Synthesis based on True-concurrency Models Source. Proc. IFIP 14th International Conf. on Testing Communicating Systems XIV, 301-316 (2002)

30. Ulrich, A., Chanson, S.: An approach to testing distributed software systems. Proc. Fifteenth IFIP WG6.1 Int. Symposium on Protocol Specification, Testing and Verification XV, 121 – 136 (1995)
31. Ulrich, A., König, H.: Specification-based Testing of Concurrent Systems. Formal Description Techniques and Protocol Specification, Testing and Verification, 17. T. Mizuno, N. Shiratori, T. Higashino & A. Togashi (Eds.) (1997)
32. Y.-S. Ma, J. Offutt and Y.-R. Kwon. μJava: An Automated Class Mutation System, Journal of Soft. Testing, Verif. and Reliability, 15(2):97-133, http://ise.gmu.edu/~ofut/mujava/ (2005)
33. Yang, R.D., Chung, C.G.: A Path Analysis Approach to Concurrent Program Testing. Information and Software Technology, 34(1):43-56 (1992)
34. Yang, C., Souter, A.L., Pollock, L.L.: All-du-path Coverage for Parallel Programs. International Symposium on Software Testing and Analysis, 153-162 (1998)
35. Kim, M.C., Chanson, S.T., Kang, S.W., Shin, J.W.: An approach for testing asynchronous communicating systems. 9th Int'l Workshop on Testing of Communicating Systems (1996)
36. Lynch N.A., Tuttle, M.R.: An introduction to Input/Output Automata. CWI Quarterly, 2(3):219–246 (1989)
37. Phalippou, M.; Executable testers. In Omar Ra.q, editor, Proceedings of the 6th International Workshop on Protocol Test Systems (IWPTS 1993), volume C-19 of IFIP Transactions, 35–50 (1994)
38. Van Der Bijl, M., Rensink, A., Tretmans J.: Compositional Testing with IOCO. FATES 2003, LNCS 2931, 86-100 (2004)
39. Gotzhein, R., Khendek, F.: Compositional Testing of Communication Systems. TestCom 2006, LNCS 3964, 227-244 (2006)

Appendix: Proof of Theorem 1

Theorem 1: Let *s* be a feasible sequence of M and s_{Li} be the projection of *s* onto ALTS L_i, $1 \leq i \leq n$. Then constrained local sequence s_{Li} is feasible for thread P_i, $1 \leq i \leq n$, iff sequence *s* is feasible for CP.
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 events executed by P_i in local sequence s_{Li}. Recall also that the local sequences of ALTS L_i are not determined by analyzing L_i and ignoring the other ALTSs in M. To the contrary, the local sequences of L_i are projections of sequences that are *definitely feasible* for M. Thus, the projected local sequences of L_i also capture the constraints that are imposed on L_i's behavior by the other ALTSs in M.

The above discussion makes the if-part of Theorem 1 obvious – 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 only-if part, the question is whether the specific thread synchronizations in sequence *s* must be feasible if all the local sequences of *s* are feasible. Suppose *e* is one of the "first" infeasible events in sequence *s*. (An event *e* is one of the first infeasible events in *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 could potentially affect e_2 [15]). There are three cases in which *e* 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*. In order for *e* to be infeasible, 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 the send partner *e'* for *e* cannot synchronize with *e*. But this contradicts the assumption that *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 *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. □

	Global model (states/trans)	#Totally-ordered seqs.	#Partially-ordered seqs.	Largest TIM (states/trans.)	#Modular seqs.
TDME	192 / 348	67,894	30	68 / 90	33
DP-3	76 / 126	238	6	8 / 8	9
DP-4	322 / 712	94,526	14	8 / 8	12
DP-5	1364 / 3770	108,549,484	30	8 / 8	15
DP-6	5778 / 19164	217,113,360,382	62	8 / 8	18
DME-3	367,733 / 1,403,821	> 3.5 trillion	4,032	71 / 117	315

Table I. Results of modular test generation.