Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs

Richard H. Carver and Kuo-Chung Tai, *Senior Member*, *IEEE*

Abstract—This paper presents and evaluates a specification-based methodology for testing concurrent programs. This methodology requires *sequencing constraints*, which specify restrictions on the allowed sequences of synchronization events. Sequencing constraints for a concurrent program can be derived from the program's formal or informal specification. Details of the proposed testing methodology based on the use of Constraints on Succeeding and Preceding Events (CSPE) are given. How to *achieve coverage* and *detect violations* of CSPE constraints for a concurrent program, according to *deterministic* and *nondeterministic testing* of this program, are described. A coverage criterion for CSPE-based testing is defined and analyzed. The results of empirical studies of CSPE-based testing for four concurrent problems are reported. These results indicate that the use of sequencing constraints for specification-based testing of concurrent programs is a promising approach.

Index Terms—Software testing, specification-based testing, concurrent programs, sequencing constraints.

1 Introduction

Two basic approaches to software testing are *specification-based* and *program-based* testing. Research on testing sequential software indicates that both approaches are needed for effective fault detection [1], [2]. Program-based testing of concurrent programs was studied in [3], [4], [5], [6], [7]. Previous work on specification-based testing of concurrent programs can be grouped into two areas, based on the type of model used to specify program behavior.

Significant progress has been made using the finite state machine model for specification-based testing of communication protocols [8], [9]. Here, valid program behavior is modeled as the sequence of events that may be observed during the execution of a conforming implementation of the finite-state machine. Conformance tests are event sequences that are selected to verify that an implementation contains the same states and transitions as its specification. Several problems can be encountered during conformance testing: inconclusive test results may be produced due to nondeterminism in the implementation; state explosion can occur when constructing finite state models; and test explosion may result when using coverage criteria that require tests for every transition, especially when illegal transitions are considered. Guidance is needed for testing a subset of the transitions when the total number of transitions is too high.

Another approach to specifying program behavior is to generate a list of required properties or "constraints." *Sequencing constraints* for a concurrent program P are used to specify restrictions on the event sequences of P. Informal examples of constraints are:

- During an execution of P, event E1 should never be followed by event E2.
- During an execution of P, after event E1, event E2 must occur before event E3.
- During an execution of P, event E1 should always be preceded by event E2.

Several notations exist for specifying constraints [10], [11], and constraints have been used for testing concurrent programs [12], [13]. During nondeterministic executions of program P, event sequences are collected and then analyzed to determine whether they satisfy the specified constraints. One problem with this approach is that illegal event sequences allowed by P may or may not be exercised during nondeterministic executions. Another problem is that it is impossible to use this approach to detect the existence of event sequences that are required by the constraints but not allowed by P. While many techniques exist for selecting test sequences from finite state machine models, none have been developed for using constraints to select test sequences.

This paper presents and evaluates a specification-based testing methodology that uses constraints to guide the selection of test sequences. Section 2 describes two basic approaches to testing concurrent programs, called *nondeterministic* and *deterministic testing*. Section 3 introduces a constraint notation called CSPE (*C*onstraints on *S*ucceeding and *P*receding *E*vents) and presents our specification-based testing methodology. This methodology uses sequencing constraints and a combination of nondeterministic and deterministic testing. The next three sections address issues involved in applying our testing methodology with CSPE constraints. Section 4 shows how to achieve coverage and detect violations of CSPE constraints. Section 5 defines a coverage criterion for CSPE-based testing and analyzes the

- *R.H. Carver is with the Department of Computer Science, George Mason University, Fairfax, VA 22030. E-mail: rcarver@cs.gmu.edu.*
- *K.-C. Tai is with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695. E-mail: kct@csc.ncsu.edu.*

Manuscript received 26 July 1996; revised 26 Aug. 1997.
Recommended for acceptance by M.L. Soffa.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105556.

effectiveness of this criterion. Section 6 presents the empirical results of applying CSPE-based testing to four concurrent programming problems. In Section 7, we compare our testing approach to related work in specification-based testing. As we will see, our methodology addresses the problems caused by nondeterminism. Furthermore, it can be applied with a combination of finite-state and constraint-based testing methods. CSPE constraints can be automatically derived from finite-state models, and test sequences can be automatically generated to cover the constraints. The use of CSPE to specify constraints has several advantages for dealing with state and test explosion during test selection. Section 8 concludes this paper.

2 Basic Approaches to Testing Concurrent Programs

In this section, we define a type of fault, called a synchronization fault, and briefly describe two approaches to testing concurrent programs. Both approaches can be used in combination with sequencing constraints. More details about these and other testing approaches can be found in [14].

2.1 Synchronization Faults

Let P be a concurrent program. An execution of P nondeterministically exercises a sequence of synchronization events (or *SYN-events*), called a *synchronization sequence* (or *SYN-sequence*). The format of a SYN-sequence depends upon the concurrent programming constructs used in P. If P is, for example, a concurrent Ada program, then a SYN-sequence of P is a sequence of rendezvous events, which includes information about the calling and called tasks for each rendezvous, the entry name for the rendezvous, etc. Formal definitions of SYN-sequences for various constructs and languages were given in [14], [15], [16].

The result of an execution of P, which includes P's output and termination condition, is determined by P and the input and SYN-sequence of this execution. Due to the unpredictable rate of progress of concurrent processes and the use of nondeterministic programming constructs, repeated executions of P with the same input may exercise different SYN-sequences and produce different results. A SYN-sequence S is said to be *feasible* (*valid*) for P with input X if S is allowed by the implementation (specification) of P with input X; otherwise, S is said to be *infeasible* (*invalid*) for P with input X. That is, the validity of a SYN-sequence of P is based on P's specification, while the feasibility of a SYN-sequence of P is based on P's implementation. P is said to have a *synchronization fault* if a feasible (valid) SYN-sequence of P with input X is invalid (infeasible).

2.2 Two Approaches to Testing Concurrent Programs

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

1) Select a set of inputs of P,
2) For each selected input X, execute P with X *many* times and examine the result of each execution. If the SYN-sequence of each execution of P can be collected, then determine the validity of each collected SYN-sequence.

The selection of inputs can be based on the specification and/or implementation of P. Multiple, nondeterministic executions of P with input X may exercise different feasible SYN-sequences and thus may detect more faults than a single execution of P with input X. However, nondeterministic testing of P with input X has three major problems. First, some feasible SYN-sequence of P with input X may be exercised many times, which is inefficient. Second, some feasible SYN-sequences of P with input X may never be exercised. Third, the existence of SYN-sequences that are valid, but infeasible for P with input X can never be detected.

Deterministic testing of P involves the following steps:

1) Select a set of tests, each of the form (X, S), where X and S are an input and a SYN-sequence of P, respectively.
2) For each selected test (X, S):
   - force a *deterministic execution* of P with input X according to S. (S is feasible for P with input X if and only if the forced execution exercises exactly S before P terminates.)
   - compare the actual and expected results (including the output, feasibility of S, and termination condition) of the forced execution.

Deterministic testing of P has the following advantages:

- It allows the use of selected SYN-sequences to test specific portions or paths of P or to satisfy a specific test criterion for P.
- It can detect the existence of feasible, invalid SYN-sequences of P, as well as valid, infeasible SYN-sequences of P with a given input.
- After P has been modified for correction or enhancement, deterministic (regression) testing with the inputs and SYN-sequences of previous executions of P provides more confidence about the correctness of P than nondeterministic testing of P.

Forced executions can also be used to repeat a feasible SYN-sequence of P with a given input; this is commonly referred to as *program replay* [17].

A test for deterministic testing consists of an input and a SYN-sequence of P, and is referred to as an *IN-SYN test*. The selection of IN-SYN tests can be based on the specification and/or implementation of P. In [3], [4] the selection of paths of a concurrent program was studied, where a path defines a SYN-sequence and a set of inputs. (A totally-ordered path of P defines a totally-ordered SYN-sequence of P and a set of inputs of P, which are the inputs in the "input domain" of this path. More discussion on the relationship between paths and SYN-sequences of a concurrent program is given in [14].) In this paper, we consider specification-based selection of SYN-sequences. This selection is based on validity constraints, which specify restrictions on the valid SYN-sequences. Note that the selection of a SYN-sequence also requires the selection of inputs associated with this SYN-sequence. For the sake of simplicity, we often use "SYN-sequences" instead of "SYN-sequences and their associated inputs" or "paths" in our discussion.

Deterministic testing of P requires a forced execution of P according to an IN-SYN test. This can be done by modifying

the implementation of the synchronization constructs used in P [17], controlling the run-time scheduler or debugger during an execution of P, or applying a source transformation strategy to P [15], [16]. The source transformation strategy is to first identify the types of statements in P that execute SYN-events. Then, immediately before (after) each such statement in P, insert an additional statement to request (release) a permit, referred to as a *SYN-permit*, for the event. (Inserted statements use the same set of synchronization constructs as used in P.) Requests for SYN-permits are granted at run-time in the order specified by a given SYN-sequence. Let P' be P transformed for deterministic testing. A SYN-sequence S is feasible for P with input X if and only if an execution of P' with (X, S) as input exercises exactly S before P' terminates. Since the problem of determining whether a program terminates is, in general, undecidable, a practical way to deal with this problem is to set a maximum allowed interval between two consecutive SYN-events, referred to as a *SYN-event interval*. If an execution of P' with (X,S) as input results in a violation of the SYN-event interval (i.e., there is no occurrence of a SYN-event within the SYN-event interval), then P' is forced to terminate abnormally, and S is assumed to be infeasible. The source transformation strategy for determining SYN-sequence feasibility is independent of the implementation of synchronization constructs in a concurrent program and does not create a portability problem.

3 Constraint-Based Testing of Concurrent Programs

In this section, we first define a constraint notation called CSPE. CSPE constraints can be specified manually, or they can be automatically derived from finite state machine specifications. We then present a constraint-based methodology for testing concurrent programs. This methodology uses both nondeterministic and deterministic testing.

3.1 Constraints on Succeeding and Preceding Events

The Constraints on Succeeding and Preceding Events (CSPE) notation was defined in [18] for the purpose of describing *feasibility constraints*, which are the sequencing constraints that are satisfied by the feasible SYN-sequences of a concurrent program. We have extended this notation to define *validity constraints*, which are the constraints that are expected to be satisfied by the feasible SYN-sequences of a concurrent program. Validity constraints are based on an interleaving model of concurrency, with constraints placed on totally-ordered sequences of events.

Two types of SYN-sequences are used to define validity constraints: a SYN-sequence S is said to be *prefix-valid* for P with input X if S is a prefix of a valid SYN-sequence of P with input X; otherwise, S is said to be *prefix-invalid* for P with input X. We will denote a SYN-sequence by using two or more events connected by a dot "." symbol, e.g., E1.E2.

We now define three basic types of CSPE constraints on succeeding events. A CSPE constraint involves two events and is preceded by a constraint operator, which is one of the following: "a" (for *always*), "~" (for *never*), and "p" (for *possibly*). Let E1 and E2 denote SYN-events and U a set of concurrent processes. Informally,

- $a[E1 ; \rightarrow E2]_U$ denotes that during an execution of U, *immediately after* an occurrence of event E1, an occurrence of event E2 is *always* valid. Thus, for any SYN-sequence v, if v.E1 is prefix-valid for U with some input, then v.E1.E2 is prefix-valid for U with the same input.
- $\sim[E1 ; \rightarrow E2]_U$ denotes that during an execution of U, *immediately after* an occurrence of event E1, an occurrence of event E2 is *never* valid. Thus, there exists no SYN-sequence v such that v.E1.E2 is prefix-valid for U with some input.
- $p[E1 ; \rightarrow E2]_U$ denotes that during an execution of U, *immediately after* an occurrence of event E1, an occurrence of event E2 is *possibly* valid (i.e., neither always valid nor never valid). Whether event E1 can be followed immediately by event E2 depends on the input and the events preceding E1. Thus, there exists an input X and a SYN-sequence v such that v.E1.E2 is prefix-valid for U with input X. Also, there exists an input Y and a SYN-sequence w such that w.E1.E2 is prefix-invalid for U with input Y. (In terms of the other constraints, $p[E1 ; \rightarrow E2]_U \equiv$ not $(a[E1 ; \rightarrow E2]_U$ *or* $\sim[E1 ; \rightarrow E2]_U)$.)

In the remainder of this paper, E1 may be "#," which denotes the start of an execution, E2 may be "$," which denotes the normal termination of an execution, and unit name U is omitted if it is implied.

CSPE constraints can be formally defined as logical formulas in the propositional modal μ-calculus and then automatically derived from finite state model using model checking tools [19]. μ-calculus definitions for CSPE are presented in Appendix A.

A constraint of the form $p[E1 ; \rightarrow E2]_U$ implies that the validity of executing E2 after E1 depends on the satisfaction of some condition. To make this condition explicit, we allow $p[E1 ; \rightarrow E2]_U$ to be extended to $p[E1 ; \rightarrow E2]_U C$, where C is a predicate that indicates the necessary and sufficient condition under which it is valid for event E1 to be immediately followed by event E2. (Condition C is called the "pre-condition" of event E2.) As an example, assume that $p[E1 ; \rightarrow E2]_U C$ is specified and v.E1 is prefix-valid for U with input X. Then immediately after the execution of U with input X and SYN-sequence v.E1, C is expected to be true if and only if v.E1.E2 is prefix-valid for U with input X. We assume that neither "True" nor "False" is used as condition C, since $a[E1 ; \rightarrow E2]_U$ can be viewed as $p[E1 ; \rightarrow E2]_U$True, and $\sim[E1 ; \rightarrow E2]_U$ can be viewed as $p[E1 ; \rightarrow E2]_U$False. Details about the use of preconditions during nondeterministic and deterministic testing will be given later.

The bounded buffer problem will be used to illustrate CSPE constraints. A bounded buffer is used to buffer communications between producers and consumers. Producers input data items and deposit them in the buffer. Consumers withdraw items from the buffer and output them. We will use a bounded buffer of size n. A finite state machine for the bounded buffer is shown in Fig. 1. (The "#" and "$" symbols do not appear in Fig. 1.)

CSPE validity constraints for an n-slot bounded buffer BB are shown below. The FSM in Fig. 1 satisfies these constraints.

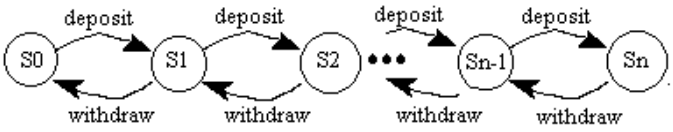

Fig. 1. FSM for an n-slot bounded buffer.

B1. a[# ;→ deposit]: Buffer BB can always accept *deposit* first. This is apparent in Fig. 1 where state S0 has an outgoing transition for *deposit*.

B2. ~[# ;→ withdraw]: Buffer BB cannot accept *withdraw* first. In Fig. 1, state S0 has no outgoing transition for *withdraw*.

B3. a[deposit ;→ withdraw]: After any *deposit*, it is always valid for the buffer to accept *withdraw*. In Fig. 1, all the states with an incoming *deposit* transition also have an outgoing *withdraw* transition.

B4. p[deposit ;→ deposit] (the buffer is not full): After a *deposit*, if and only if the buffer is not full, it is valid to accept another *deposit*. In Fig. 1, constraint a[deposit ;→ deposit] does not hold since state Sn, in which the buffer is full, has an incoming *deposit* transition but no outgoing *deposit* transition. Constraint ~[deposit ;→ deposit] does not hold since there are states with both incoming and outgoing *deposit* transitions.

B5. a[withdraw ;→ deposit]: After any *withdraw*, it is always valid for the buffer to accept *deposit*. In Fig. 1, all the states with an incoming *withdraw* transition have an outgoing *deposit* transition.

B6. p[withdraw ;→ withdraw](the buffer is not empty): After a *withdraw*, if the buffer is not empty, it is valid to accept another *withdraw*.

CSPE can be used to specify constraints on valid SYN-sequences, but a program's correctness depends on both the validity of its SYN-sequences and the correctness of its outputs. For example, an execution of a FIFO bounded buffer is correct if it exercises a valid sequence of deposits and withdraws, and the data items are withdrawn in the order that they are deposited. Constraints (B1)–(B6) above do not specify the FIFO ordering property. It is possible, however, to specify such constraints, if inputs and outputs are modeled as SYN-events between the system and its environment, and we use a new type of CSPE constraint called an abstract constraint.

3.2 Abstract CSPE Constraints

To make CSPE constraints more expressive and flexible, we allow them to be written for a user-defined subset of the program events, called the *observable events*. Events that are not observable are called *hidden events*. An abstract CSPE constraint references observable events only.

This type of abstraction is not new. A distinction between observable and hidden events is used in process algebras to prove that two systems have equivalent observable behavior. The selection of observable event sequences for *black-box* conformance testing of communication protocols has been studied for over a decade. Our motivation for using this abstraction in CSPE is that we can now specify sequencing constraints for events that do not necessarily appear consecutively in totally-ordered SYN-sequences. This allows us to focus on a selected subset of events, and the properties involving those events, by hiding the other events. Also, as we will see later, abstract constraints support the use of reduction techniques for avoiding state explosion when constructing finite state machine models and generating test sequences.

Let E1 and E2 denote observable SYN-events and U a set of concurrent processes. An abstract validity constraint expresses a restriction on observable event E1 being succeeded immediately by observable event E2 in a totally-ordered SYN-sequence of U. (Zero, one, or more hidden events may occur between E1 and E2.) Abstract constraints are of the same three types as defined earlier: after an occurrence of E1, an occurrence of E2 is either *always*, *never*, or *possibly* valid. However, the constraints must be defined in a way that captures the effects of hidden events on observable behavior. In the following constraint definitions, α is a (possibly empty) sequence of observable and hidden events in U, and y is a (possibly empty) sequence of hidden events in U. An abstract constraint on E1 being succeeded by E2 is denoted by $[[E1 ;\rightarrow E2]]_U$. Informally,

- $a[[E1 ; \rightarrow E2]]_U$: For any SYN-sequence α, if α.E1 is prefix-valid for U with some input, then there exists a y such that α.E1.y.E2 is prefix-valid for U with the same input.
- $\sim[[E1 ; \rightarrow E2]]_U$: There exist no SYN-sequences α and y such that α.E1.y.E2 is prefix-valid for U with some input.
- $p[[E1 ; \rightarrow E2]]_U$: After valid sequence α.E1 is exercised, an occurrence of observable event E2 is *possibly* valid (i.e., neither always nor never valid). Whether event E1 can be followed by E2 depends on the input and events preceding E1 and hidden events that occur after E1. (In terms of the other constraints, $p[[E1 ; \rightarrow E2]]_U \equiv$ not ($a[[E1 ; \rightarrow E2]]_U$ *or* $\sim[[E1 ; \rightarrow E2]]_U$).)

As with nonabstract constraints, we allow $p[[E1 ; \rightarrow E2]]_U$ to be extended with a precondition to $p[[E1 ; \rightarrow E2]]_U C$. When there are no hidden events in U, the constraints $[[E1 ; \rightarrow E2]]_U$ and $[E1 ; \rightarrow E2]_U$ are equivalent. Formal definitions of abstract CSPE constraints in the propositional modal μ-calculus are given in Appendix A.

To illustrate abstract constraints, consider another version of the bounded buffer. Assume that buffer BB-h is a 2-slot buffer formed by composing two 1-slot buffers that communicate via a hidden *handoff* event, as shown in Fig. 2a. A handoff event indicates the transfer of one item from the left 1-slot buffer (which accepts the deposit for the item) to the right 1-slot buffer (which accepts the withdraw for the item). A finite state machine for BB-h is shown in Fig. 2b.

The abstract constraints for buffer BB-h, with observable events *deposit* and *withdraw* and hidden event *handoff*, are given below. The FSM in Fig. 2b satisfies these constraints.

B7. a[[# ; → deposit]]. In Fig. 2b, state S0 has an outgoing transition, which is for observable event *deposit*.

B8. ~[[# ; → withdraw]]. In Fig. 2b, state S0 has neither an outgoing *withdraw* transition nor a hidden *handoff* transition that takes it to a state with an outgoing *withdraw* transition.

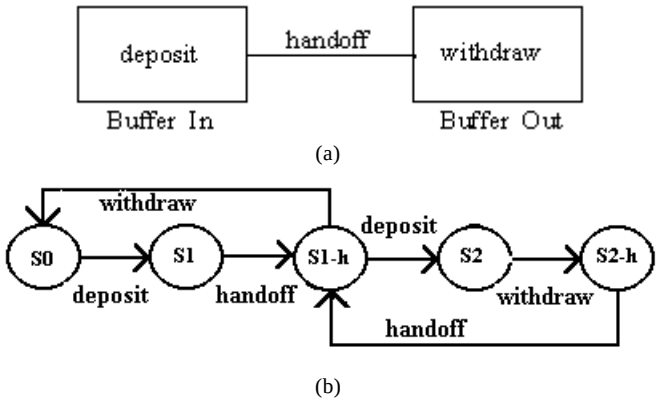

Fig. 2. (a). A 2-slot bounded buffer BB-h as a composition of two 1-slot buffers; (b) FSM for buffer BB-h.

B9. a[[deposit ;→ withdraw]]. In Fig. 2b, all the states with an incoming *deposit* transition have either an outgoing *withdraw* transition, or they have a hidden *handoff* transition that takes them to a state having an outgoing *withdraw* transition.

B10. p[[deposit ; → deposit]] (the buffer is not full). Constraint a[[deposit ; → deposit]] does not hold since state S2, in which the buffer is full, has an incoming *deposit* transition but it has neither an outgoing *deposit* transition nor a hidden *handoff* transition that takes it to a state with an outgoing *deposit* transition. Constraint ~[[deposit ; → deposit]] does not hold since there are states that have an incoming and outgoing *deposit* transition, or an incoming *deposit* transition and a hidden *handoff* transition to a state with an outgoing *deposit* transition.

B11. a[[withdraw ; → deposit]]. Similar to B9.

B12. p[[withdraw ; → withdraw]] (the buffer is not empty). Similar to B10.

Constraints B7 through B12 are exactly the same as constraints B1 through B6, respectively, except that the former are abstract constraints and the latter are not. The use of the hidden event *handoff* does not change the observable behavior of the buffer, which is reflected in the resulting constraints.

As another example, we specify the FIFO property for the bounded buffer. Assume that the Producer inputs data items I1, I2, and I3 and deposits them into buffer BB-h in that order, while the Consumer withdraws these items in FIFO order and outputs them as O1, O2, and O3. Abstract constraints for BB-h, with observable events I1, I2, and I3, are given below:

B13. a[[# ;→ I1]]	**B16**. a[[I1 ;→ I2]]	**B19**. a[[I2 ;→ I3]]
B14. ~[[# ; → I2]]	**B17**. ~[[I1 ;→ I1]]	**B20**. ~[[I2 ; → I2]]
B15. ~[[# ; → I3]]	**B18**. ~[[I1 ; → I3]]	**B21**. ~[[I2 ; → I1]]

Abstract constraints for BB-h with observable events O1, O2, and O3 are obtained by replacing each input event in constraints B13 through B21 with the corresponding output event. The resulting CSPE constraints capture the requirement for FIFO ordering of data items in BB-h. Notice that input events I1, I2, and I3 never occur consecutively in a totally-ordered SYN-sequence of BB-h, nor do events O1, O2, and O3, but abstract constraints allow us to express the required FIFO behavior.

3.3 A Constraint-Based Methodology for Testing Concurrent Programs

Our proposed methodology for testing a concurrent program P consists of the following steps:

1) Select a sequencing constraint notation, define how to achieve coverage and detect violations of constraints written in this notation, and determine a constraint coverage criterion to be used as the goal of testing.
2) According to P's specification, use the selected constraint notation to derive a set of validity constraints. If P's specification is formal, then constraints may be derived automatically.
3) Perform nondeterministic testing of P and collect the exercised SYN-sequences. Then analyze the collected SYN-sequences to measure coverage and detect violations of P's validity constraints.
4) Generate additional SYN-sequences of P with the intention of using these SYN-sequences to cover the validity constraints that have not yet been covered.
5) Perform deterministic testing of P with the generated SYN-sequences. (If deterministic testing of P with a generated SYN-sequence fails to cover a constraint, then a constraint violation is detected.)
6) After corrections have been made to P, perform deterministic (regression) testing with the SYN-sequences that previously uncovered violations. By doing so, we can make sure that the detected faults have been fixed. We may also perform deterministic testing with additional SYN-sequences to make sure that no new faults have been introduced into P.

The above testing methodology involves both deterministic and nondeterministic testing and does not depend upon any particular notation for specifying validity constraints. The following three sections show the application of the above testing methodology based on CSPE. Section 4 shows how to achieve coverage and detect violations of CSPE constraints, according to deterministic and nondeterministic testing. Section 5 defines a coverage criterion for CSPE-based testing and analyzes the effectiveness of this coverage criterion for fault detection. Section 6 presents the empirical results of applying CSPE-based testing to four concurrent programming problems. The use of other constraint notations with the above testing methodology is discussed in Section 7.

4 Coverage and Violations of CSPE Constraints

The first step of our constraint-based testing methodology is to define how to achieve coverage and detect violations of constraints. Section 4.1 shows rules about how a CSPE constraint for a concurrent program P is *covered* or *violated* by a feasible SYN-sequence of P. It also shows how these rules are applied during nondeterministic testing. Section 4.2 gives details about achieving coverage and detecting violations of CSPE constraints when deterministic testing is used.

Before we present the rules for test generation, we must first consider the relation that we expect to hold between the SYN-sequences in P and those in P's specification. This

relation affects the SYN-sequences that can be selected during specification-based testing. We use the following two sets of SYN-sequences to define some possible relations between P and its specification:

Feasible(P, X) = the set of feasible SYN-sequences of P with input X

Valid(P, X) = the set of valid SYN-sequences of P with input X

A number of implementation relations have been proposed:

1) Feasible(P, X) is a proper subset of Valid(P, X). This relation is used when the specification of P uses nondeterminism to model open implementation decisions. Implementation decisions are made and then reflected in P, which amounts to a *reduction* of the nondeterminism in the specification. If more precision is needed, a specification may distinguish between necessary and admissible behavior of the implementation [20].
2) Valid(P, X) is a proper subset of Feasible(P, X). This relation is used when the specification of P is incomplete. Thus, P is an *extension* of its specification.
3) Feasible(P, X) = Valid(P, X). This is the relation we use in this paper. Every event sequence allowed by P's specification must be allowed by P's implementation, and vice versa.

Relations 1) and 2) allow some flexibility when implementing a specification. However, during deterministic testing, there must be no question about whether SYN-sequences selected from the specification are expected to be feasible or infeasible for the implementation; otherwise, the results of the tests will be inconclusive. We assume that relation 3) is required to hold and thus that a specification written in CSPE (or one from which CSPE constraints are derived) does not leave open any design decisions that may affect these constraints.

Based on relation 3), P is said to be *correct for input X* if and only if

- Feasible(P, X) = Valid(P, X), and
- Every execution of P with input X produces the correct or expected result.

P is said to be *correct* if and only if P is correct with every possible input. Notice that checking whether Feasible(P, X) = Valid(P, X) does not completely answer the question of correctness. During deterministic testing, the results (or output) of an execution must also be checked.

Relations 1) and 2) are sometimes appropriate, and our methodology does not exclude the use of these relations during development. For example, a specification S1 can be written using nondeterminism to model open design decisions. When the design decisions are made, they are reflected in implementation S2. A reduction relation 1) can be used to verify implementation S2 using some other verification technique (e.g., [20]). The new implementation S2 then becomes the specification for the next implementation S3, and the implementation relation is changed to 3). Implementation S3 can be verified using deterministic testing against specification S2.

The restriction on using reduction relations is for deterministic testing; when CSPE constraints are used during nondeterministic testing, a reduction relation is allowed. To see why, assume that implementation P is a reduction of its specification. Thus, some SYN-sequences allowed by P's specification are not allowed by P. This relation has no effect on nondeterministic testing since nondeterministic testing cannot detect a reduction. (It is impossible to use nondeterministic testing to show that a valid SYN-sequence is not allowed by P.)

4.1 Rules for Coverage and Violations of CSPE Constraints

We consider a CSPE constraint for P to be *covered* if the constraint is satisfied at least once during some execution of P. Thus, the coverage of a CSPE constraint for P does not imply that this constraint will be satisfied by every execution of P. If a constraint does not hold for all possible executions of P, then the constraint may be covered by some executions of P and violated by others, or it may be covered by some portion of an execution of P and violated by another portion of the same execution.

The rules in this section are based on abstract CSPE constraints. For these rules, E1 and E2 denote observable SYN-events, α and β denote sequences of hidden and observable SYN-events, and y denotes a (possibly empty) sequence of hidden events only. Rules based on nonabstract CSPE constraints are special cases of those based on abstract CSPE constraints, since they assume that no hidden events exist (i.e., y is empty); these rules were shown in [21]. We will illustrate the rules with SYN-sequences of program BB-h, which implements a 2-slot bounded buffer by composing two 1-slot buffers (see Fig. 2). Program BB-h has observable events *deposit* and *withdraw* and a hidden *handoff* event.

For the rules below, we define two new types of SYN-sequences: a SYN-sequence S is said to be *prefix-feasible* for P with input X if S is a prefix of a feasible SYN-sequence of P with input X; otherwise, S is said to be *prefix-infeasible* for P with input X.

Rules for Coverage and Violations

1) Constraint a[[E1 ;→ E2]] is *covered* by SYN-sequence α.E1.y.E2 and input X, provided that α.E1.y.E2 is prefix-valid and prefix-feasible for P with input X. For example, constraint a[[deposit ; → withdraw]] is covered by SYN-sequence #.deposit.handoff.withdraw if this sequence is prefix-feasible for program BB-h.
2) Constraint a[[E1 ;→ E2]] is *violated* by SYN-sequence α.E1.y.E2 and input X, provided that α.E1.y is prefix-feasible and α.E1.y.E2 is prefix-valid, but prefix-infeasible, for P with input X. For example, constraint a[[deposit ; → withdraw]] is violated by SYN-sequence #.deposit.handoff.withdraw if #.deposit.handoff is prefix-feasible for program BB-h, but #.deposit.handoff. withdraw is not.
3) Constraint ~[[E1 ; → E2]] is *covered* by SYN-sequence α.E1.y.E2 and input X, provided that α.E1.y is prefix-valid and prefix-feasible for P with input X, but α.E1.y.E2 is prefix-infeasible. For example, constraint ~[[# ; → withdraw]] is covered by SYN-sequence

#.withdraw if this sequence is prefix-infeasible for program BB-h.

4) Constraint ~[[E1 ;→ E2]] is *violated* by SYN-sequence α.E1.y.E2 and input X, provided that α.E1.y is prefix-valid and α.E1.y.E2 is prefix-feasible for P with input X. For example, constraint ~[[# ; → withdraw]] is violated by SYN-sequence #.withdraw if this sequence is prefix-feasible for program BB-h.
5) The *true* situation of constraint p[[E1 ; → E2]]C is *covered* by SYN-sequence α.E1.y.E2 and input X, provided that α.E1.y.E2 is prefix-valid and prefix-feasible for P with input X. (Since α.E1.y.E2 is prefix-valid for P with input X, immediately after the execution of P with X and α.E1.y, the value of C is expected to be true.) For example, the true situation of constraint p[[deposit ; → deposit]](the buffer is not full) is covered by SYN-sequence #deposit.handoff.deposit if this sequence is prefix-feasible for program BB-h.
6) The *false* situation of constraint p[[E1 ; → E2]]C is *covered* by SYN-sequence α.E1.y.E2 and input X, provided that
   - α.E1.y is prefix-valid for P with input X, but α.E1. y.E2 is not, and
   - α.E1.y is prefix-feasible for P with input X, but α. E1.y.E2 is not.

   (Since α.E1.y.E2 is prefix-invalid for P with input X, immediately after the execution of P with X and α.E1.y, the value of C is expected to be false.) For example, the false situation of constraint p[[deposit ; → deposit]](the buffer is not full) is covered by SYN-sequence #.deposit.handoff.deposit.deposit if #.deposit.handoff.deposit is prefix-feasible for program BB-h but #.deposit.handoff.deposit.deposit is not.
7) Constraint p[[E1 ; → E2]]C is *violated* by SYN-sequence α.E1.y.E2 and input X provided that α.E1.y is both prefix-feasible and prefix-valid for P with input X and one of the following two conditions holds:

   a) α.E1.y.E2 is prefix-feasible, but prefix-invalid for P with input X.
   b) α.E1.y.E2 is prefix-infeasible, but prefix-valid for P with input X.

   For example, constraint p[[deposit ; → deposit]] (the buffer is not full) is violated by SYN-sequence #.deposit.handoff.deposit.deposit, if it is prefix-feasible for program BB-h (due to 7a)), and by SYN-sequence #.deposit.handoff.deposit, if it is prefix-infeasible for program BB-h (due to 7b)).

Rules 1), 4) 5), and 7a) are satisfied only by prefix-feasible SYN-sequences of P. Since all the SYN-sequences collected during nondeterministic testing are feasible, they can be analyzed to achieve *coverage* of "always" constraints (rule 1)) and true situations of "possibly" constraints (rule 5)); and to detect *violations* of "never" constraints (rule 4)), and "possibly" constraints (rule 7a)). However, there is no guarantee that by applying nondeterministic testing to P, we will cover all the constraints that are in theory coverable by nondeterministic testing and detect all the constraint violations that are in theory detectable by nondeterministic testing.

As an example, assume that SYN-sequences S1, S2, and S3 shown below were collected during executions of BB-h. For each of these SYN-sequences, we give the CSPE constraints that it covers or violates:

- Sequence S1 is #.deposit.handoff.withdraw.deposit. handoff.deposit.$. It covers a[[# ;→ deposit]], a[[deposit ; → withdraw]], a[[withdraw ; → deposit]], and the true situation of p[[deposit ; → deposit]].
- Sequence S2 is #.deposit.handoff.deposit.deposit.$. It covers a[[# ; → deposit]] and the true situation of p[[deposit ; → deposit]], and violates p[[deposit ; → deposit]].
- Sequence S3 is #.deposit.handoff.withdraw.withdraw. It covers a[[# ;→ deposit]] and a[[deposit ; → withdraw]], and violates p[[withdraw ; → withdraw]].

Rules 2), 3), 6), and 7b) are satisfied only by prefix-infeasible SYN-sequences of P. Since prefix-infeasible SYN-sequences of P can never be exercised during nondeterministic testing of P, deterministic testing of P is needed to achieve *coverage* of "never" constraints (rule 3)) and false situations of "possibly" constraints (rule 6)); and to detect *violations* of "always" constraints (rule 2)) and "possibly" constraints (rule 7b)). Also, deterministic testing of P can accomplish whatever nondeterministic testing of P can in terms of covering constraints and detecting constraint violations. The next section gives details about how to achieve coverage and detecting violations of CSPE constraints by using deterministic testing.

4.2 Using Deterministic Testing to Achieve Coverage and Detect Violations of CSPE Constraints

Deterministic testing of P is used to force executions of selected SYN-sequences to cover specific CSPE constraints. In an attempt to cover a CSPE constraint for P, we select a SYN-sequence and perform deterministic testing of P according to this SYN-sequence. This testing has three possible outcomes: 1) the constraint is covered, 2) the constraint is violated, or 3) a fault is detected which prevents the execution from reaching the position in the SYN-sequence where the coverage is attempted. If 3) occurs, the detected fault in P needs to be fixed before the next attempt to cover the constraint. Below we provide rules for selecting SYN-sequences and determining the outcome of an attempt to cover a CSPE constraint.

We first define a successful outcome of a forced execution: a forced execution of P with input X and SYN-sequence S is said to *succeed* for a prefix S* of S if the forced execution does not terminate before the end of S* is reached (i.e., S* is prefix-feasible for P with input X).

Rules for Deterministic Testing

8) To cover constraint a[[E1 ;→ E2]] or the true situation of constraint p[[E1 ; → E2]]C, we select SYN-sequence α.E1.y.E2.β and input X such that α.E1.y.E2 is prefix-valid for P with input X. For a forced execution of P with X and α.E1.y.E2.β :
   - if it does not succeed for α.E1.y, then a fault is detected in P,

- if it succeeds for α.E1.y, but not for α.E1.y.E2, then the constraint is violated,
- if it succeeds for α.E1.y.E2, then the constraint is covered.

(For p[[E1 ;$\rightarrow$ E2]]C, the value of C is expected to be true immediately before the expected execution of E2.) As an example, for program BB-h, SYN-sequence #.deposit.handoff.withdraw.$ can be used in an attempt to cover a[[# ; $\rightarrow$ deposit]] and a[[deposit ; $\rightarrow$ withdraw]]. SYN-sequence #.deposit.handoff.deposit.withdraw.handoff.withdraw.$ can be used in an attempt to cover the true situations of p[[deposit ; $\rightarrow$ deposit]] and p[[withdraw ; $\rightarrow$ withdraw]].

9) To cover constraint ~[[E1 ;$\rightarrow$ E2]] or the false situation of constraint p[[E1 ;$\rightarrow$ E2]]C, we select SYN-sequence α.E1.y.E2 and input X such that α.E1.y is prefix-valid for P with input X, but α.E1.y.E2 is not. For a forced execution of P with X and α.E1.y.E2:

   - if it does not succeed for α.E1.y, then a fault is detected in P,
   - if it succeeds for α.E1.y, but not for α.E1.y.E2, then the constraint is covered,
   - if it succeeds for α.E1.y.E2, then the constraint is violated.

   (This rule uses α.E1.y.E2 instead of α.E1.y.E2.β, since α.E1.y.E2 is expected to be prefix-infeasible for P with input X. For p[[E1 ;$\rightarrow$ E2]]C, the value of C is expected to be false immediately before the attempted execution of E2.) As an example, for program BB-h SYN-sequence #.withdraw can be used in an attempt to cover ~[[# ;$\rightarrow$ withdraw]]. And SYN-sequence #.deposit.handoff.withdraw.withdraw can be used in an attempt to cover the false situation of p[[withdraw ; $\rightarrow$ withdraw]].

Note that in Rule 8), α.E1.y.E2 is prefix-valid for P with input X, and is expected to be prefix-feasible, while in Rule 9), α.E1.y.E2 is prefix-invalid for P with input X and is expected to be prefix-infeasible. A valid SYN-sequence of P with input X can be used to cover "always" constraints and the true situations of "possibly" constraints, but such a sequence cannot be used to cover "never" constraints and the false situations of "possibly" constraints. As shown in Rule 9), an invalid SYN-sequence of P with input X is needed to cover (using the very last event in this SYN-sequence) a "never" constraint or the false situation of a "possibly" constraint. Such a SYN-sequence may also cover "always" constraints and the true situations of "possibly" constraints before the end event is reached; this will reduce the number of sequences needed to achieve coverage of the constraints.

Based on the number and types of constraints, it is possible to determine the minimum number of test sequences that are need to cover the constraints. If all the constraints are "always" constraints, then a minimum of one test sequence is needed. Otherwise, the minimum number of test sequences is the number of "never" constraints plus the number of "possibly" constraints. (One or more of these sequences can be used to cover all "always" constraints and all true situations of "possibly" constraints.)

If a constraint is covered by a forced execution, outputs must be checked to determine the correctness of this execution. Also, the selection of a SYN-sequence requires the selection of input data associated with this SYN-sequence. Many concurrent programs are data-independent, that is, their SYN-sequences do not depend on their input data. For these programs, the selection of inputs is not linked to the selection of SYN-sequences. For data-dependent programs, selecting input data and SYN-sequences from informal specifications is difficult. When formal specifications are used, however, this selection can be partially automated. Many algorithms have been developed for selecting input data from finite state machine specifications during conformance testing [8], [9]. As we show in Appendix A, CSPE constraints can be formally defined in temporal logic. This allows CSPE constraints and the SYN-sequences that cover the constraints to be generated automatically from finite state machine specifications. If the specifications model input events as SYN-events between the system and its environment, SYN-sequences and their associated input data can be generated together. The problem then becomes one of controlling the potential state explosion during the construction of the state machine. Abstract CSPE constraints support existing techniques for deriving reduced machine models—input events can be hidden and the resulting state machine can be reduced using incremental analysis algorithms that collapse semantically equivalent states into a single state without affecting the observable behavior. This means that CSPE constraints can be derived from reduced models, and SYN-sequences and their associated inputs can be selected from reduced models [22], [23].

To summarize, in this section we have discussed above how the rules for coverage and violations of CSPE constraints can be applied during nondeterministic and deterministic testing of P. Deterministic testing can achieve 100 percent coverage of P's constraints. Thus, deterministic testing is more effective than nondeterministic testing for providing constraint coverage and detecting constraint violations. However, since deterministic testing requires more programmer effort than nondeterministic testing, it is more cost-effective to apply nondeterministic testing first, and then apply deterministic testing to cover the uncovered constraints.

5 A CSPE-Based Test Coverage Criterion

Below we define a CSPE-based test coverage criterion. We first illustrate this criterion by applying it to the 2-slot bounded buffer problem. We then analyze the theoretical effectiveness of this criterion for fault detection.

Definition. *The* CSPE-1 coverage criterion *for a concurrent program P requires that 1) each "always" or "never" validity constraint for P be covered at least once, and 2) the true and false situations of each "possibly" validity constraint for P be covered at least once.*

Following our earlier discussion, the CSPE-1 criterion can never be satisfied by using nondeterministic testing only; it requires either deterministic testing or a combination of nondeterministic and deterministic testing.

5.1 Application of the CSPE-1 Criterion to the 2-Slot Bounded Buffer Problem

Table 1 shows four SYN-sequences that can be used to satisfy the CSPE-1 criterion for a 2-slot bounded buffer using deterministic testing. Let BB be a program that implements a 2-slot bounded buffer to satisfy constraints B1 through B6. (Program BB has two observable events *deposit* and *withdraw*, and no hidden events.) The constraints intended to be covered by each sequence are indicated in column two. The true situations of B4 and B6 are denoted as (B4)t and (B6)t, respectively. Similarly, the false situations of B4 and B6 are denoted as (B4)f and (B6)f, respectively. (Since S5, S6, and S7 are prefix-invalid, if any of them is prefix-feasible for an implementation of the 2-slot bounded buffer problem, then a fault is detected.)

TABLE 1
SYN-SEQUENCES FOR THE BOUNDED BUFFER PROBLEM

SYN-sequences	Constraints Intended to Cover
S4: #.deposit.deposit.withdraw.withdraw	B1, (B4)t, B3, (B6)t
S5: #.withdraw	B2
S6: #.deposit.deposit.deposit	B1, (B4)t, (B4)f
S7: #.deposit.withdraw.deposit.withdraw.withdraw	B1, B3, B5, (B6)f

The valid SYN-sequences of the 2-slot bounded buffer problem can be defined by the FSM (finite state machine) M1 shown in Fig. 3, with S0 as the start state and S0, S1, and S2 as final states:

A commonly used test generation criterion for an FSM is *transition coverage*, which is to cover every transition in the FSM at least once. The transition coverage criterion for M1 can be satisfied by S4, which covers only four of eight constraints. The FSM M2 shown in Fig. 4 extends M1 by adding an error state Error and two transitions to state Error for invalid inputs. The transition coverage criterion for M2 can be satisfied by {S4, S5, S6}, which does not cover constraints B5 and (B6)f. Thus, for the 2-slot bounded buffer problem, the CSPE-1 criterion is stronger than the transition coverage criterion.

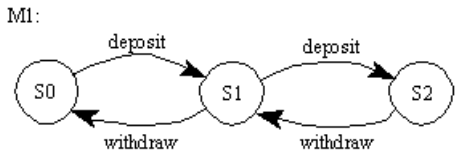

Fig. 3. An FSM defining SYN-sequences for a 2-slot bounded buffer.

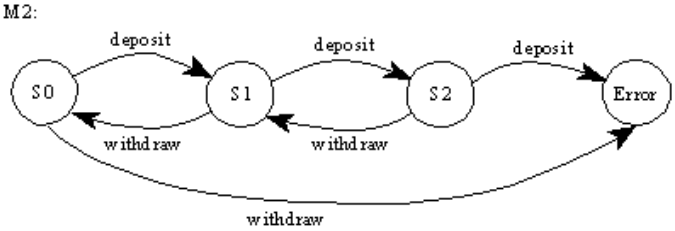

Fig.4. Extension of FSM M1 to include an error state.

5.2 Fault Detection by the CSPE-1 Criterion and Nondeterministic Testing

The CSPE validity constraints for a concurrent program P are expected to be satisfied by all possible executions of P. Let the *CSPE feasibility constraints* for P be the CSPE constraints that are actually satisfied by P. A "mismatch" between a CSPE validity constraint for P and its corresponding feasibility constraint indicates the existence of faults in P. Table 2 shows the possible types of mismatches (or faults) between CSPE validity and feasibility constraints without hidden events. The table indicates whether a mismatch will be detected by the CSPE-1 criterion. For example, the "a-n" mismatch in row one refers to the situation that for events E1 and E2 in P, a[E1; → E2] holds according to the specification of P, but ~[E1;→ E2] holds according to the implementation of P. The table also indicates whether a mismatch will be detected by the *NT-max coverage criterion*, which refers to 100 percent coverage of the feasibility constraints that are in theory coverable by nondeterministic testing.

TABLE 2.
DETECTION OF CSPE FAULTS BY CSPE-1 AND NT-MAX WITHOUT HIDDEN EVENTS

mis-match	validity constraint	feasibility constraint	detected by CSPE-1	detected by NT-max
a-n	a[E1;→ E2]	~[E1 ;→ E2]	yes (rule (8))	no
a-p	a[E1;→ E2]	p[E1;→ E2]	maybe	no
n-a	~[E1 ;→ E2]	a[E1;→ E2]	yes (rule 9))	yes (rule 4))
n-p	~[E1 ;→ E2]	p[E1 ;→ E2]	maybe	yes (rule 4))
p-n	p[E1 ;→ E2]	~[E1 ;→ E2]	yes (rule 8))	no
p-a	p[E1 ;→ E2]	a[E1 ;→ E2]	yes (rule 9))	maybe
p-p-d	p[E1;→ E2]C	p[E1;→ E2]¬C	yes (rule 8) or 9))	yes (rule 7a))
p-p-i	p[E1 ;→ E2]C	p[E1 ;→ E2]C*	maybe	maybe

Below is an explanation of Table 2. In the following discussion, α and y denote SYN-sequences.

- Row one indicates that an "a-n" mismatch will definitely be detected by the CSPE-1 criterion, but will not be detected by the NT-max criterion. The reason is that the intended coverage of a[E1 ;→ E2] by deterministic testing according to rule 8) definitely produces a violation, while nondeterministic testing can never detect the mismatch.
- Row two indicates that an "a-p" mismatch may be detected by the CSPE-1 criterion. The reason is that when SYN-sequence α.E1.E2 and input X are used in an attempt to cover a[E1 ;→ E2], where α.E1 is both prefix-valid and prefix-feasible for P with input X, a violation of a[E1 ;→ E2] is detected only if α.E1.E2 is prefix-infeasible for P with input X.
- Row four, which is for an "n-p" mismatch, is the only row in which the CSPE-1 criterion has a weaker fault detection capability than the NT-max criterion. The reason is that when SYN-sequence α.E1.E2 and input X are used in an attempt to cover ~[E1;→ E2], where α.E1 is both prefix-valid and prefix-feasible for P with

input X, a violation of ~[E1 ;→ E2] is detected only if α.E1.E2 is prefix-feasible for P with input X. However, the coverage of the true situation of p[E1 ;→ E2] by nondeterministic testing definitely detects a violation.

- The "p-p-d" or "p-p-disjoint" mismatch in row seven refers to the situation that the pre-condition of E2 in the validity constraint p[E1 ;→ E2] is the complement of the precondition of E2 in the feasibility constraint p[E1 ;→ E2]. This mismatch can be detected by deterministic testing according to either rule (8) or (9), which attempts to cover the true or false situation of the validity constraint p[E1 ;→ E2]. Also, this mismatch can be detected by nondeterministic testing when the true situation of the feasibility constraint p[E1 ;→ E2] is covered.
- The "p-p-i" or "p-p-intersection" mismatch in row eight refers to the situation that for validity constraint p[E1 ;→ E2]C and feasibility constraint p[E1 ;→ E2]C*, C and C* are neither equivalent nor disjoint.

From Table 2, we have the following observations:

- The CSPE-1 criterion guarantees the detection of "a-n", "n-a", "p-n", "p-a", and "p-p-d" mismatches, while the NT-max criterion does so only for "n-a", "n-p", and "p-p-d" mismatches.
- The NT-max criterion can never detect an "a-n" or "a-p" or "p-n" mismatch. For a validity constraint a[E1 ;→ E2], nondeterministic testing can never detect any mismatch ("a-n" or "a-p"), since a violation of this constraint requires an infeasible SYN-sequence (rule 2)). Also, the detection of an "p-n" mismatch requires an infeasible SYN-sequence (rule 7b)).
- The CSPE-1 criterion has stronger or equivalent fault detection capability than the NT-max criterion except for an "n-p" match.
- To improve the detection of "a-p", "n-p" and "p-p-i" mismatches, more stringent requirements for the coverage of "possibly" constraints can be considered. Since a "possibly" constraint is associated with a predicate (i.e., its necessary and sufficient condition), predicate testing strategies (e.g., [35], [36], [37], [38]) can be applied to define various coverage criteria for a "possibly" constraint. (Note that the CSPE-1 criterion requires only *branch coverage* of the predicate associated with a "possibly" constraint.) More stringent coverage criteria for "possibly" constraints can improve the effectiveness of CSPE-based testing of concurrent programs.

Now we consider the detection of CSPE faults by CSPE-1 and NT-max when some events are hidden. The entries in Table 2 are still valid except that the entries corresponding to column "detected by CSPE-1" and rows "n-a" and "p-a" are changed from "yes" to "maybe." Below we explain the difference for "n-a." Assume that SYN-sequence α.E1.E2 and input X of program P are used in an attempt to cover validity constraint ~[E1 ;→ E2], where α.E1 is prefix-valid for P with input X, but α.E1.E2 is not, and that a[E1 ;→ E2] is the feasibility constraint. If deterministic testing of P with input X and α.E1.E2 succeeds for α.E1, then it succeeds for α.E1.E2 and thus a violation is detected. On the other hand, assume that SYN-sequence α.E1.y.E2 and input X of program P are used in an attempt to cover validity constraint ~[[E1 ;→ E2]], where α.E1.y is prefix-valid for P with input X, but α.E1.y.E2 is not, and that a[[E1 ;→ E2]] is the feasibility constraint. If deterministic testing of P with input X and α.E1.y.E2 succeeds for α.E1.y, it may succeed or fail for α.E1.y.E2 and thus there is no guarantee that a violation is detected.

6 Empirical Studies of CSPE-Based Testing of Concurrent Programs

This section shows some results of our empirical studies of CSPE-based testing. Our objective was to investigate the following two questions:

1) What is the effectiveness of CSPE constraints for fault detection during nondeterministic testing?
2) What is the effectiveness of CSPE constraints and the CSPE-1 criterion for fault detection during deterministic testing? (Deterministic testing can be combined with nondeterministic testing to satisfy the CSPE-1 criterion. However, nondeterministic testing by itself can never satisfy the CSPE-1 criterion.)

Our empirical studies involved the use of correct and incorrect Ada programs for solving several concurrent problems, including the bounded buffer, readers and writers, gas station, and sliding window protocol problems. A source transformation tool (see Section 2), called TDCAda [16], was used to perform nondeterministic and deterministic testing.

6.1 CSPE-Based Testing for the Bounded Buffer Problem

To answer question 1), a 2-slot bounded buffer was implemented in Ada. Let this program be referred to as BB. Program BB contains three tasks: Producer, Consumer, and Buffer_Control. Task Producer (Consumer) calls entry deposit (withdraw) of task Buffer_Control two times to deposit (withdraw) two items. A set of 120 nonequivalent mutants of program BB was generated by using the Mothra mutation system [24]. (Since Mothra was developed for Fortran 77, a Fortran version of program BB was used to generate mutants, and then the Ada versions of these mutants were produced.) Delay statements with randomly assigned delay times were inserted into the BB mutants in order to increase the chance of executing different SYN-sequences during nondeterministic testing. For each BB mutant, the mutant was executed to collect one SYN-sequence, and then the collected SYN-sequence was analyzed to detect violations of constraints B1 through B6 shown in Section 3.1. (The SYN-sequences in this and the other studies were collected automatically using the source transformation tools in TDCAda.). Each SYN-sequence contained the two deposits and two withdraws exercised during the execution of program BB. If no violation was found, then the BB mutant was executed again to collect one more SYN-sequence, and so on. This process was continued until one violation was found. The results were as follows:

- For 101 BB mutants, a constraint violation was detected in the first collected SYN-sequence.

- For eight BB mutants, no constraint violation was found until the fourth collected SYN-sequence.
- For eight BB mutants, no constraint violation was found until the 16th collected SYN-sequence.
- For the remaining three BB mutants, no constraint violation could be found by nondeterministic testing. The reason was that each of these three mutants implemented a 1-slot buffer and thus had only one feasible SYN-sequence: #.deposit.withdraw.deposit.withdraw.deposit.withdraw, which was a valid sequence.

These results show that the CSPE constraints for BB are very effective for detecting faults in BB mutants when SYN-sequences are collected during nondeterministic testing. Also, the existence of three mutants with no constraint violations during nondeterministic testing points out the need for deterministic testing.

To answer question 2), we used SYN-sequences S4 through S7 (shown in Section 5) to perform deterministic testing of each of the 120 BB mutants. Notice that the set {S4, S5, S6, S7} satisfies the CSPE-1 criterion, but any proper subset of {S4, S5, S6, S7} does not. A BB mutant was considered to be killed if a forced execution of the mutant according to a prefix-valid (prefix-invalid) SYN-sequence for BB was found to be prefix-infeasible (prefix-feasible), or the forced execution produced an incorrect result. Below are the numbers of mutants killed by each of S4 through S7 and by selected subsets of {S4, S5, S6, S7}.

	Number of mutants killed by subsets of sequences S4, S5, S6, and S7						
subset	S4	S5	S6	S7	{S4, S5}	{S4, S5, S6}	{S4, S5, S6, S7}
mutants killed	103	12	50	101	105	114	120

The results show that {S4, S5, S6, S7} kills all 120 BB mutants. Thus, CSPE constraints and the CSPE-1 criterion provided 100 percent fault detection during deterministic testing of a 2-slot buffer program.

6.2 CSPE-Based Testing for the Readers and Writers Problem

In the readers and writers (RW) problem, data is shared by several processes. When a process reads (writes) the shared data, it is considered to be a reader (writer). Readers may access the shared data concurrently, but a writer always has exclusive access. Different strategies can be used to control how readers and writers access the shared data. In Table 3, we describe seven different RW strategies based on different priority schemes. These seven RW strategies can be classified into the following three categories:

1) R = W: readers and writers have an equal priority and are served in FIFO (first-in-first-out) order of their requests to read or write.
2) R > W: readers generally have a higher priority than writers.
3) R < W: readers generally have a lower priority than writers.

For R > W and R < W, readers are served in FIFO order and writers are served in FIFO order. Within each category, different strategies exist due to changing priorities in specific situations. The difference between strategies R < W.1 and R < W.2 is the following. Assume that when writer W1 requests to write, a reader R1 is reading and another reader R2 has requested to read but has not starting reading. R < W.1 lets W1 start before R2, while R < W.2 forces W1 to start after R2. For R > W.1 and R > W.2, writers starve if before a reader finishes reading, the next reader arrives. For R < W.1 and R < W.2, readers starve if before a writer finishes writing, the next writer arrives. For R = W.1, R = W.2, and R < W.3, no readers or writers starve.

TABLE 3
STRATEGIES FOR THE READERS AND WRITERS PROBLEM

R = W.1	one reader or writer with equal priority.
R = W.2	many readers or one writer with equal priority.
R > W.1	many readers or one writer with readers having a higher priority.
R > W.2	same as R > W.1 except that at the end of a writing, if both readers and writers are waiting, writers that requested to write before the first waiting reader requested to read have a higher priority.
R < W.1	many readers or one writer with writers having a higher priority.
R < W.2	same as R < W.1 except that when a writer arrives, if no writer is writing or waiting, it waits until all readers that arrived earlier have finished.
R < W.3	same as R < W.1 except that at the end of a writing, waiting readers have a higher priority than waiting writers.

To derive CSPE constraints from the informal specifications of the various RW strategies in Table 3, we need to make some assumptions about the designs of the solutions for these strategies. We assume that

- each reader performs start_read (SR), reads shared data, and then performs end_read (ER),
- each writer performs start_write (SW), writes shared data, and then performs end_write (EW).
- each SR (SW) event is preceded with a request for read (write).

When deriving CSPE constraints, we consider a request for read or write as a hidden SYN-event and SR, ER, SW, and EW as observable SYN-events. These observable events were chosen because they mark the starts and ends of the critical sections. If we consider a request for read or write as an observable SYN-event, then the number of CSPE constraints for an RW strategy would increase significantly.

Since each CSPE constraint involves two SYN-events and the RW problem has four observable SYN-events, there are sixteen different CSPE constraints for each RW strategy. In addition, we need to include four constraints with "#" as the first event and four constraints with "$" as the second event. Thus, each RW strategy has a total of 24 constraints. The following CSPE constraints are true for each of the RW strategies in Table 3.

The seven RW strategies differ in only eight CSPE constraints. These eight constraints are shown in Table 4 for the seven strategies. No two of the seven strategies have the same set of CSPE constraints.

CSPE constraints			
(1)	~[[# ;→ ER]]	(9)	~[[SW ;→ SW]]
(2)	~[[# ; → EW]]	(10)	a[[SW ;→ EW]]
(3)	a[[SR ;→ ER]]	(11)	~[[EW ;→ ER]]
(4)	~[[SR ;→ SW]]	(12)	~[[EW ;→ EW]]
(5)	~[[SR ;→ EW]]	(13)	~[[SR ;→ $]]
(6)	~[[ER ;→ EW]]	(14)	p[[ER ;→ $]] (no readers are reading)
(7)	~[[SW ;→ SR]]	(15)	~[[SW ;→ $]]
(8)	~[[SW ;→ ER]]	(16)	a[[EW ;→ $]]

Since the differences between the seven RW strategies are very subtle, it is possibly that when a program is intended to implement one of the seven RW strategies, it actually implements a different strategy. In other words, a program that correctly implements one of the seven RW strategies can be viewed as a "mutant" of another program that correctly implements a different strategy. To carry out our empirical studies, one Ada program was written to correctly implement each of the seven different RW strategies. Each program contained three readers and two writers, each of which performed one read or write operation.

Question 1) in our empirical studies was to investigate the effectiveness of fault detection by using CSPE constraints during nondeterministic testing. To answer this question for the RW problem, we investigated the effectiveness of fault detection by nondeterministic testing, with the assumption that each of the seven Ada programs had each of the other six RW strategies as its specification. The empirical study was done in steps. First, each of the seven Ada programs was executed one hundred times to collect one hundred SYN-sequences. Second, the collected SYN-sequences of the program for one RW strategy were analyzed, using the CSPE validity constraints for each of the other six RW strategies, to determine when the first constraint violation occurred. The results of this analysis are shown in Table 5. An entry in Table 5 that contains an integer, say k, and has strategy A as row and strategy B as column means that the first constraint violation was detected in the kth collected SYN-sequence of the program that had strategy A as its specification but actually implemented strategy B. (Entries on the diagonal, indicated by "–", were not considered in the analysis.) Only one nondiagonal entry is blank, which corresponds to the situation in which the program had strategy R = W.2 as its specification but actually implemented strategy R = W.1. This entry must be blank since no such violation can be detected by nondeterministic testing. The results in Table 5 indicate that CSPE constraints for the RW problem are very effective for fault detection during nondeterministic testing.

Question 2) in our empirical studies was to investigate the effectiveness of fault detection by using CSPE constraints and the CSPE-1 criterion during deterministic testing. To answer this question for the RW problem, we investigated the effectiveness of fault detection by the CSPE-1 criterion, with the assumption that each of these seven Ada programs had each of the other six RW strategies as its specification. For each of the seven RW strategies, we manually constructed one test set of SYN-sequences to satisfy the CSPE-1 criterion for that strategy's CSPE constraints. A test set contained four to eight test sequences. We used the shortest and minimum number of sequences (see Section 4.2) that covered the strategy's constraints without considering the other strategies. Each SYN-sequence consisted of requests, starts, and ends of read and write operations, and the values to be written by write operations. We then used the test set for each RW strategy to perform deterministic testing of the programs implementing the other six RW strategies. When a SYN-sequence in a test set was used during deterministic testing, the values obtained from read operations and the feasibility of the SYN-sequence were examined. According to the results of the empirical study, the test set for a given RW strategy was able to distinguish the implementation for this strategy from the other six implementations. Thus, the CSPE-1 criterion for the RW problem is very effective for fault detection during deterministic testing.

6.3 CSPE-Based Testing for the Gas Station Problem

The gas station problem [13] is an often used example in concurrent programming. In the gas station problem, each customer must prepay the operator for a selected pump. Upon receiving payment from a customer, if the selected pump is not being used, the operator activates the pump. Customers selecting the same pump must use the pump in the order of prepayment. To use a pump, a customer first waits for the activation of the pump by the operator, starts pumping, finishes pumping, and then receives change from the operator. After a customer finishes pumping, the amount to charge the customer is sent from the pump to the operator. Upon receiving a charge from a pump, the operator activates the pump if a customer is waiting for the pump, and determines the amount of change to give to the customer who just finished using the pump.

To derive validity constraints from the above informal specification of the gas station problem, we need to make some assumptions about the design of a solution to this problem. Here we assume that the following SYN-events are involved: prepay for a pump, activate a pump, start using a pump, end using a pump, charge an amount to a customer, and give change to a customer. Table 6 shows the names and parameters of each SYN-event. In the following discussion, C_i and C_j, $0 < i, j <= m$ denote customers and P_k and P_r, $0 < k, r <= n$, pumps, where m and n are the numbers of customers and pumps, respectively.

In Table 6, to ensure that customers selecting the same pump use the pump in the order of prepayment, C_i is a parameter of both A_P and S_P. When a customer tries to perform S_P, the customer identification in S_P is compared to that in A_P, which was passed from the operator. If these two identifications do not match, the customer is not allowed to use the pump.

Each customer C_i, $i > 0$, performs the following sequence of SYN-events:

$$P\_P(C_i, P_k), S\_P(C_i, P_k), E\_P(P_k), CN(C_i).$$

Each pump P_k, $k > 0$, performs the following sequence of SYN-events for each customer:

$$A\_P(C_i, P_k), S\_P(C_i, P_k), E\_P(P_k), CR(C_i, P_k).$$

TABLE 4

(a) R = W.1			
p[[# ;→ SR]]	the reader is first in fifo order	p[[# ;→ SW]]	the writer is first in fifo order
~[[SR ;→ SR]]		p[[ER ;→ SW]]	the writer is next in fifo order
p[[ER ;→ SR]]	the 2nd reader is next in fifo order	p[[EW ;→ SR]]	the reader is next in fifo order
~[[ER ;→ ER]]		p[[EW ;→ SW]]	the 2nd writer is next in fifo order

(b) R = W.2			
p[[# ;→ SR]]	the reader is first in fifo order	p[[# ;→ SW]]	the writer is first in fifo order
p[[SR ;→ SR]]	the 2nd reader is next in fifo order	p[[ER ;→ SW]]	no reader is reading and the writer is next in fifo order
p[[ER ;→ SR]]	the 2nd reader is next in fifo order	p[[EW ;→ SR]]	the reader is next in fifo order
p[[ER ;→ ER]]	both readers read concurrently	p[[EW ;→ SW]]	the 2nd writer is next in fifo order

(c) R > W.1			
a[[# ;→ SR]]		p[[# ;→ SW]]	no reader is waiting
a[[SR ;→ SR]]		p[[ER ;→ SW]]	no reader is reading or waiting
a[[ER ;→ SR]]		a[[EW ;→ SR]]	
p[[ER ;→ ER]]	both readers read concurrently	p[[EW ;→ SW]]	no reader is waiting

(d) R > W.2			
a[[# ;→ SR]]		p[[# ;→ SW]]	no reader is waiting
a[[SR ;→ SR]]		p[[ER ;→ SW]]	no reader is reading or waiting
a[[ER ;→ SR]]		p[[EW ;→ SR]]	no waiting writer requested before the reader
p[[ER ;→ ER]]	both readers read concurrently	p[[EW ;→ SW]]	no waiting reader requested before the writer

(e) R < W.1			
p[[# ;→ SR]]	no writer is waiting	a[[# ;→ SW]]	
p[[SR ;→ SR]]	no writer is waiting	p[[ER ;→ SW]]	no reader is reading
p[[ER ;→ SR]]	no writer is waiting	p[[EW ;→ SR]]	no writer is waiting
p[[ER ;→ ER]]	both readers read concurrently	a[[EW ;→ SW]]	

(f) R < W.2			
	no writer is waiting	a[[# ;→ SW]]	
p[[# ;→ SR]]	no writer was waiting	p[[ER ;→ SW]]	all readers that requested earlier than the writer have finished reading
p[[ER ;→ SR]]	when the 2nd reader requested, no writer was waiting	p[[EW ;→ SR]]	no writer is waiting or (when the writer finished and when the reader requested, no writer was waiting)
p[[E+R ;→ ER]]	both readers read concurrently	p[[EW ;→ SW]]	when the 2nd writer requested, either the 1st writer was writing or (the 1st writer finished writing and there was no waiting readers)

(g) R < W.3			
p[[# ;→ SR]]	no writer is waiting	a[[# ;→ SW]]	
p[[SR ;→ SR]]	no writer is waiting or both readers requested before the most recent writer finished	p[[ER ;→ SW]]	no reader is reading
p[[ER ;→ SR]]	no writer is waiting	p[[EW ;→ SR]]	the reader requested before the writer finished or no writer is waiting
p[[ER ;→ ER]]	both readers read concurrently	p[[EW ;→ SW]]	no waiting readers requested before the first writer finished

TABLE 5
NUMBER OF SEQUENCES UNTIL THE FIRST VIOLATION DETECTED

	R = W.1	R = W.2	R > W.1	R > W.2	R < W.1	R < W.2	R < W.3
R = W.1	–	9	1	1	2	1	1
R = W.2		–	1	3	2	1	1
R > W.1	1	1	–	2	1	1	2
R > W.2	3	2	1	–	2	1	1
R < W.1	3	2	1	3	–	31	1
R < W.2	3	2	1	3	31	–	1
R < W.3	1	1	1	2	1	1	-

TABLE 6
DEFINITIONS OF SYN-EVENTS IN THE GAS STATION PROBLEM

SYN-event	Name and Parameters	SYN-event	Name and Parameters
Pre_Pay	P_P(C_i, P_k)	End_Pumping	E_P(P_k)
Activate_Pump	A_P(C_i, P_k)	Charge	CR(C_i, P_k)
Start_Pumping	S_P(C_i, P_k)	Change	CN(C_i)

Once a pump starts A_P for one customer, it completes the above sequence before starting A_P for another customer. A set of CSPE validity constraints for the gas station problem is shown in Table 7. Table 7a contains one constraint for two customers and Table 7b 22 constraints for the operator. Note that the constraint in Table 7a only deals with two start_pumping events of two customers and considers other types of events as hidden events. Also, the constraints in Table 7b deal with only the types of events that involve the operator and consider other types of events as hidden events. By doing so, more constraint information can be expressed in the CSPE notation. Constraints for each customer and pump, which reflect the simple SYN-sequences shown above, are straightforward and have been omitted.

In [13], four incorrect solutions to the gas station problem were shown, and a deadlock was found in each of the first three solutions by applying nondeterministic testing. In [25], it was shown that the fourth solution also has a deadlock and that the deadlock in the first solution can be detected by using one customer, one pump, and the operator, while deadlocks in other solutions can be detected by using two or three customers, one pump, and the operator. The deadlocks in the first and second solutions violate constraint (5b), while deadlocks in the third and fourth solutions result from executions violating constraint (1), which specifies the first-pay-first-use rule for a pump.

We investigated the effectiveness of using CSPE constraints to detect faults in these implementations during nondeterministic testing. Each of the four solutions was executed to collect SYN-sequences until the first constraint violation was detected. Table 8 shows the results. An entry in Table 8 that contains an integer, say k, and has solution A as the column means that the first constraint violation was detected in the kth collected SYN-sequence of solution A. The constraints violated were constraints (5b) and (1), as described above. The results in Table 8 indicate that CSPE constraints for the gas station problem are very effective for fault detection during nondeterministic testing. An analysis of the gas station solutions showed that there can be executions that violate the constraints but do not have a deadlock, while every execution with a deadlock has a constraint violation. Thus, it is more effective to check for violations of the constraints than to wait until deadlocks are detected.

6.4 CSPE-Based Testing for the Sliding Window Protocol

The Sliding Window Protocol (SWP) supports a unidirectional flow of messages with a positive handshake on each transfer, and an acknowledgment window for flow control. It is an important flow-control and error recovery technique, which is present in many real protocols. For this empirical study, we considered a version of SWP in which messages are sent tagged with circular sequence numbers in the range 1..6, and these sequence numbers also constitute the acknowledgments. The size of the acknowledgment window was two.

We developed a formal specification of SWP in Lotos [26] and derived from this specification a set of CSPE constraints and a set of test sequences that satisfied the CSPE-1 criterion for these constraints. The constraints were derived automatically by using the temporal logic definitions of CSPE constraints given in the Appendix. The test sequences were automatically generated using the test generation tool described in [27]. This tool inputs (1) a program A written in Lotos, and (2) a set of CSPE validity constraints derived from A. The tool outputs a set of SYN-sequences that cover A's constraints. In addition to the test sequences required for CSPE-1 coverage, special-value sequences were generated by applying the CSPE constraints specifically to sequence numbers 6-1. Based on our experience, this "wrap-around" case was considered to be a likely source of faults in the implementation. This process resulted in a total of 210 test sequences with an average length of 36 Ada rendezvous events.

We then developed an Ada implementation of the SWP that had six tasks and 242 statements. To measure the adequacy of the generated test sequences, this Ada SWP implementation was semiautomatically mutated using mutation operators of the Mothra mutation system. The result was a set of 1,009 nonequivalent mutants. We used the 210 test sequences to perform deterministic testing of each of the 1,009 SWP mutants. The 210 sequences distinguished

TABLE 7

(a) Constraint for Two Customers C_i and C_j, i/=j	
(1)	p[[S_P(C_i, P_k) ;→ S_P(C_j, P_k)]] (P_P(C_i,P_k) precedes P_P(C_j,P_k)).

(b) Operator Constraints (only P_P, A_P, CR, and CN involve the operator)		
(2a)	a[[# ;→ P_P(C_i, P_k)]]	
(2b)	~[[# ; → A_P(C_i, P_k)]]	
(2c)	~[[# ; → CR(C_i, P_k)]]	
(2d)	~[[# ; → CN(C_i)]]	
(3a)	p[[P_P(C_i, P_k) ; → A_P(C_j, P_r)]]	i = j, k = r, and P_k is not being used
(3b)	p[[P_P(C_i, P_k) ; → P_P(C_j, P_r)]]	i/=j and P_k is being used
(3c)	p[[P_P(C_i, P_k) ; → CR(C_j, P_r)]]	i/=j and P_k and P_r are being used (k and r may be the same)
(3d)	~[[P_P(C_i, P_k) ; → CN(C_j)]]	
(4a)	p[[A_P(C_i, P_k) ; → PP(C_j, P_r)]]	i/=j
(4b)	a[[A_P(C_i, P_k) ; → CR(C_i, P_k)]]	
(4c)	p[[A_P(C_i, P_k) ; → CR(C_j, P_r)]]	i/=j, k/=r, and P_r is being used.
(4d)	p[[A_P(C_i, P_k) ; → CN(C_j)]]	i/=j, C_j just finished using P_k, and P_k has waiting customers
(4e)	~[[A_P(C_i, P_k) ; → A_P(C_j, P_r)]]	
(5a)	p[[CR(C_i, P_k) ; → A_P(C_j, P_k)]]	i/=j and P_k has waiting customers.
(5b)	p[[CR(C_i, P_k) ; → CN(C_j)]]	i = j and P_k has no waiting customers
(5c)	~[[CR(C_i, P_k) ; → P_P(C_j, P_r)]]	
(5d)	~[[CR(C_i, P_k) ; → CR(C_i, P_r)]]	
(6a)	a[[CN(C_i) ; → P_P(C_j, P_k)]]	
(6b)	p[[CN(C_i) ;→ CR(C_j, P_k)]]	P_k is being used
(6c)	~[[CN(C_i) ; → A_P(C_j, P_k)]]	
(6d)	~[[CN(C_i) ; → CN(C_j)]]	
(7)	p[[CN(C_i) ; → $]]	all other customers have received change

TABLE 8
NUMBER OF SEQUENCES UNTIL THE FIRST CONSTRAINT VIOLATION WAS DETECTED

Solution 1	Solution 2	Solution 3	Solution 4
1	1	2	2

98 percent (989/1,009) of the SWP mutants from the original SWP implementation. The results of this study indicate that a combination of finite-state and CSPE-based testing methods is effective for test generation and fault detection.

7 Related Work

In this section, we review related work in the area of specification-based testing for concurrent programs. Section 7.1 gives an overview of conformance testing using finite state machine (FSM) specifications. In Section 7.2, we describe how formal specifications have been used to analyze traces of concurrent programs.

7.1 Conformance Testing

Techniques have been developed for specification-based test generation from deterministic FSMs (DFSMs), nondeterministic FSMs (NFSMs), and labeled transition systems (LTSs). These tests, called conformance tests, try to verify that an implementation contains the same states and transitions as its specification. When inputs are selected from a single DFSM M, a test is performed by providing the implementation under test (IUT) with a sequence of inputs, called a test sequence. Test inputs are supplied by one or more test processes. The IUT passes the test only if all observed outputs match those specified by M. The general approach to selecting test sequences from M is as follows. For each transition T in M:

1) bring the IUT to a known state, e.g., the initial state, using a sequence of inputs referred to as a *synchronizing sequence*,
2) select an input sequence, called a *transferring sequence*, that brings the IUT to the head state of T,
3) check if the IUT can accept the input of transition T. If the IUT fails, a fault is detected, else proceed to step 4,
4) select an additional input sequence to verify that the tail state of T in the IUT is correct. This sequence is referred to as a *state identification sequence.*

Specific test sequence generation methods using the above general approach include the D-method, U-method, and W-method [8], [9]. After constructing test sequences for each transition, these sequences can be combined to form a single test sequence for M.

Nondeterminism is a major concern during black-box conformance testing. For a test sequence E selected from a

DFSM M, an execution of the IUT with E may be nondeterministic and produce inconclusive results because of one or both of the following reasons:

- The IUT is nondeterministic due to the presence of faults or internal events.
- If E involves two or more test processes, there is no guarantee that the IUT will accept inputs from different test processes in the order shown in E. One solution to this problem is to require synchronization of inputs between different test processes. Another solution is to use synchronizable test sequences [28].

When M is a single NFSM, the nondeterminism in the IUT creates several problems [29]: 1) it may not be possible to force the IUT into a desired state with a synchronizing or transferring sequence, and 2) it is not always possible to verify that the IUT is in a given state using the state identification sequence. A similar problem arises when testing LTSs that contain internal transitions [30]. While attempting to verify the tail state of a transition T, the IUT may choose an internal transition and end up in a state where the state identification sequence is not accepted. This does not mean that the IUT was in the wrong state after T, it means that the test sequence could not tell whether it was in the correct state or not. A technique for conformance testing of LTSs was shown in [31].

Although black-box conformance testing has problems with nondeterminism, it is appropriate for system or acceptance testing, when the source code is often inaccessible to testers or simply too large and complicated. The deterministic testing technique used in this paper addresses the problems caused by nondeterminism. It can be used in earlier phases of testing with testing tools that provide options for collecting and forcing SYN-sequences.

A second concern during conformance testing is determining whether the trace of input and output interactions observed during a test conforms to the specification. This is problematic in a distributed environment because conformance testing is a black box technique that allows only partial control and observation of the IUT. The IUT cannot be observed internally, it can only be observed at the points through which it interacts with the external environment. In a distributed system, a local observer which watches only the interactions at one of the interfaces sees only a partial trace of program execution. This affects error detection because some global event ordering errors cannot be detected by local (*external*) observers alone. This problem of observation inaccuracy with multiple testers was studied in [32], [33]. The deterministic testing technique used in this paper is not a black box technique. It provides complete, *internal* control and observation of each process in the system. Thus, error detection is not affected by observability concerns in a distributed environment.

A third concern during conformance testing is state explosion. When a system is specified as a set of communicating FSMs or LTSs, one can construct the composite machine and use it for test selection. For large systems, however, constructing a complete composite machine may be impractical due to the explosion in the number of states and transitions. Also, techniques that test every transition in the composite machine, especially those that consider invalid transitions, suffer from a test explosion problem. To cope with state explosion, *incremental testing* and *reduction* techniques can be used [22], [23]. Incremental testing involves partitioning a program into two or more components that can be tested separately. Optionally, the components may be *reduced* into smaller but semantically equivalent ones. The CSPE constraint notation supports incremental testing and reduction. Each set of observable events corresponds to a different component of program behavior. Events can be hidden in order to derive a reduced model of the program. The CSPE constraint notation also controls test explosion. In the empirical study in Section 6.4, the number of CSPE constraints derived from the composite machine for the sliding window protocol (SWP) was much less than the number of its transitions. Yet, test sequences generated from the CSPE constraints were effective at detecting errors. We emphasize that some of the SWP mutants in our study could only be killed by invalid sequences, but most existing conformance techniques generate only valid sequences. Thus, CSPE-based testing provides guidance for selecting invalid sequences while avoiding test explosion, and mutation-based testing provides a measure of the adequacy of these sequences.

7.2 Trace Analysis

Formal specifications have been used to construct trace analysis modules for nondeterministic testing. A trace analysis module reads an observed trace of the implementation and determines whether the trace is valid. If the mapping between the events in the specification and the events in the implementation is formally defined, then the validity of traces can be determined automatically.

In [32], an IUT is executed and its outputs are traced. The validity of a trace is checked by the trace analysis module, which is constructed directly from the FSM specification of the IUT. An output in the FSM specification becomes an expected input to the trace analysis module. If the sequence of outputs in the trace corresponds to a sequence of expected inputs to the trace analysis module, then the trace is valid.

A constraint notation called Task Specification Language (TSL) was described in [11]. A TSL constraint is of the form

when activating_event
then specified_event
before | **until** terminating_event

where "|" indicates a choice between keywords **before** and **until**. (In TSL, **until** means "by the time.") TSL constraints describe patterns of events that must occur or must not occur during a program's execution. For a given event sequence S, after the activating event matches an event of S:

- if the specified event matches an event of S and the terminating event matches a later event, then the TSL constraint is satisfied.
- if the terminating event matches an event of S before a match for the specified event is found, then the TSL constraint is violated.
- if the specified and terminating events match the same event of S, then
- if "**until**" is used, the TSL constraint is satisfied.

- if "**before**" is used, the TSL constraint is violated.

TSL also allows the definition of *properties*. A property is a function of the events that occur during an execution; it has an initial value that changes when specified events happen. For example, for the 2-slot buffer problem, a property called Buffer_Count can be defined with an initial value of 0. Immediately after an occurrence of event *deposit*, Buffer_Count is increased by one, and immediately after an occurrence of event *withdraw*, Buffer_Count is decreased by one. The following two TSL constraints specify the 2-slot buffer problem:

```
when Deposit where Buffer_Count = 1
  then Withdraw before Deposit;
when Withdraw where Buffer_Count = 1
  then Deposit before Withdraw;
```

Assume that a set of TSL constraints has been specified for the IUT. A SYN-sequence collected during a nondeterministic execution of the IUT can be checked to determine whether the sequence satisfies or violates each TSL constraint. If a constraint violation is detected, then the violated constraint and the SYN-sequence causing this violation are useful for debugging. Rosenblum [34] described the following TSL tools for Ada:

- The TSL / Ada compiler transforms TSL constraints and properties for an Ada program into Ada procedures that communicate with the TSL trace analysis module.
- The TSL trace analysis module collects synchronization events during an execution of an Ada program and checks for inconsistencies between the collected SYN-sequences and the TSL constraints.

[34] also showed how to use TSL to specify Ada tasking semantics and then apply the specification and the above tools to validate the behavior of a distributed Ada supervisor.

Only nondeterministic testing has been applied during TSL-based testing and debugging. However, rules could be defined for the coverage of TSL constraints during deterministic testing. For example, to apply deterministic testing to a concurrent program according to the TSL constraint "**when** E1 **then** E2 **before** E3", we could use SYN-sequence u1.E1.u2.E2.u3.E3.u4.E1.u5.E3 such that ui, $1 \leq i \leq 5$, is an event sequence that does not contain any of E1, E2, and E3, and u1.E1.u2.E2.u3.E3.u4.E1.u5 is prefix-valid for the program. This SYN-sequence is expected to satisfy the constraint once and then cause a violation at the end of the SYN-sequence.

In [10] a graphical interval logic (GIL) was defined for the specification and verification of temporal properties of concurrent systems. Specifications in GIL describe properties that must be satisfied by every sequence of states that represents an execution of the IUT. GIL formulas are evaluated over intervals, which are sequences of states that are delimited by specified events within the execution of the IUT. Dillon and Yu [12] described how to build a GIL trace analysis module for concurrent Ada programs. A trace analysis module is built by constructing an FSM that accepts precisely those state sequences satisfying a GIL formula. (An FSM is constructed for each GIL formula.) Traces are obtained by manually instrumenting the Ada source program so that relevant execution events are reported to the analysis module. Also, when a trace violates a specification, the analysis module will construct a GIL formula that describes the fault in the trace. The formula and the trace can be used to help the user debug the IUT.

TSL and GIL have only been used with nondeterministic testing. Research is needed to determine whether it is possible to develop general rules for test sequence generation that can be applied to any TSL or GIL property. When using CSPE, test generation rules exist for each type of constraint, and a set of constraints along with test sequences that cover the constraints can be automatically generated from finite state machine specifications.

8 Conclusions

Constraint-based specification and testing have a number of advantages. First, constraints can be derived from a formal or informal specification of a concurrent program. (The derived constraints become part of the specification and can be used for verification purposes.) Second, constraints can be analyzed to check for consistency and completeness. (For example, the set of CSPE constraints for a concurrent program is complete if it contains a CSPE constraint for every possible pair of events.) Third, constraints for a concurrent program can be used for testing and debugging, regardless of whether or not the constraints are complete. In this paper, we have demonstrated the effectiveness of CSPE constraint-based testing of concurrent programs.

We plan to continue our research on CSPE constraint-based testing of concurrent programs in several directions. We will investigate the use of additional types of operators to define CSPE constraints [18]. Also, we will study new coverage criteria for "possibly" constraints by applying predicate testing strategies [35], [36], [37], [38] to predicates associated with "possibly" constraints.

The constraint notations TSL and GIL have been used for specifying sequencing constraints, and for constructing trace analysis modules during nondeterministic testing. In this paper we have shown rules for coverage of CSPE constraints during deterministic and nondeterministic testing. As we showed in Section 7, similar rules can be defined for covering TSL constraints. We will consider the use of TSL, GIL, and possibly other sequencing constraint notations in theoretical and empirical studies of testing concurrent programs.

In specification-based nondeterministic testing of a concurrent program P, the events in a collected SYN-sequence S of P may need to mapped into "abstract" events used in the specification of P in order to determine the correctness of S. The definition of higher-level events in terms of lower level events and the use of such definitions in debugging concurrent programs have been studied [39], [40]. For specification-based deterministic testing of P, a specification-based SYN-sequence of P is often not an implementation-based SYN-sequence of P, due to the abstractions used in the specification and the existence of hidden events. The mapping from specification-based SYN-sequences to implementation-based SYN-sequences is a tedious task. Techniques and tools are being developed to reduce the effort required for this task [22].

APPENDIX A

In Appendix A, we describe a modal logic for labeled transition systems and define CSPE constraints in this modal logic. Processes in languages such as CCS and Lotos generate labeled transition systems of two types, depending on whether or not any events are hidden. When no events are hidden, the labeled transition system is of the form (S,{ $\rightarrow a \rightarrow \mid a \in A$ }) where S is a nonempty set of states; A is an event set; and $\rightarrow a \rightarrow$ represents the state transition resulting from the "execution" of *a*. We first define nonabstract CSPE constraints for systems with no hidden events.

The μ-calculus is a propositional modal logic interpreted over labeled transition systems. Propositions can be written to make assertions about a system, and the truth of the propositions can vary from state to state. We summarize the μ-calculus description that appears in [41]. Let K range over subsets of an event set A. The set of formulas of the logic is defined as follows:

- tt is a formula
- if P and Q are formulas, then so are $\neg$P, P $\wedge$ Q, and [K]P.

A formula is either the constant true formula tt; or a negated formula $\neg$P; or a conjunction of formulas P $\wedge$ Q; or a modalized formula [K]P whose intended meaning is: P holds after every (performance of any) event in K.

The truth of a formula in the μ-calculus is defined relative to a transition system. We use a satisfaction relation, denoted $\approx$, to define when a formula is true in a state E $\in$ S. E $\approx$ P should be read "the formula P is true in state E." The semantics of the formulas are defined as follows:

- E $\approx$ tt for all states E
- E $\approx$ P $\wedge$ Q iff E $\approx$ P $\wedge$ E $\approx$ Q
- E $\approx \neg$P iff not E $\approx$ P
- E $\approx$ [K]P iff for all *a* in K and all E', if E $\rightarrow a \rightarrow$ E' then E' $\approx$ P

Every state has the property tt. A state has the property $\neg$P when it fails to have property P, while it has property P $\wedge$ Q when it has both properties P and Q. Finally, a state satisfies [K]P if after every performance of any event in K all the resulting states have property P.

There are also derived modal operators ff $\equiv \neg$tt, P $\vee$ Q $\equiv \neg(\neg P \wedge \neg Q)$, and <K> P $\equiv \neg$ ([K] $\neg$P). The diamond operator <K> is the dual of [K]:

- E $\approx$ <K> P iff for some *a* in K and some E', E $\rightarrow a \rightarrow$ E' and E' $\approx$ P.

An important difference between the box [ ] and diamond < > operators is that [K] P holds of states that cannot perform an action in K, while <K> P holds only in states that can definitely perform an action in K.

EXAMPLE. The following formulas are true for labeled transition system D1 shown in Fig. 5. They represent either feasible (tt) or infeasible (ff) sequences:

- <E0> <E1> tt
- <E0> <E2> <E3> tt
- <E3> ff

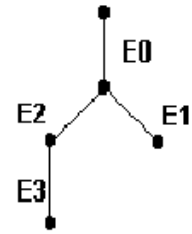

Fig. 5. LTS D1.

We will define several "macros" that are useful when defining CSPE constraints:

- <Can e> $\approx$ <e>tt
- <Can't e> $\approx \sim$ <Can e>.

Macro <Can e> (<Can't e>) expresses a capability (inability) to perform event *e*.

- Always P
- Eventually P

Always P holds in a state s if P holds in every state reachable from s. This macro is used to describe a property which invariantly holds for a labeled transition system. Eventually P holds in a state s if P holds in some state reachable from s. (Eventually P $\rightarrow \neg$(Always $\neg$P)). (Definitions of Always and Eventually in the μ-calculus require fixed-point formulas that can be found in [20].)

Table 9 shows μ-calculus definitions of nonabstract CSPE constraints.

TABLE 9
NONABSTRACT CSPE CONSTRAINTS
IN THE PROPOSITIONAL MODAL μ-CALCULUS

Constraint	μ-calculus definition
~[E1 ; $\rightarrow$ E2]	Eventually (<E1> tt) $\wedge$ Always ([E1] <Can't E2>)
a[E1 ; $\rightarrow$ E2]	Eventually (<E1> tt) $\wedge$ Always ([E1] <Can E2>)
p[E1 ; $\rightarrow$ E2]	Eventually (<E1> tt) $\wedge$ (a[E1 ; $\rightarrow$ E2] $\vee$ ~[E1 ; $\rightarrow$ E2])
a[# ; $\rightarrow$ E1]	<Can> E1

In Table 9, we use the "Eventually (<E1> tt)" clause to make sure there is at least one E1 event in the system. This is because [E1] P will always evaluate to true in systems that contain no E1 events, regardless of P. (See the above discussion about the difference between the box [ ] and diamond < > operators.)

When hidden ("τ") events exist, the transition system generated by a process is (S, {$\Rightarrow a \Rightarrow \mid a \in A$'}) where $A' = (A - \tau) \cup \varepsilon$ and $\Rightarrow a \Rightarrow$ means observable event *a* preceded by zero or more τ moves and followed by zero or more τ moves. (There is no transition relation $\Rightarrow \tau \Rightarrow$ but there is the relation $\Rightarrow \varepsilon \Rightarrow$ meaning *zero* or more τ moves. The transition E $\Rightarrow \varepsilon \Rightarrow$ E is enabled in every state E.)

To define CSPE constraints in the presence of hidden events, we use two additional modal operators [[e]] P and <<e>> P. A state satisfies [[e]] P if every e-observation derivative $\Rightarrow$ e $\Rightarrow$ satisfies P. A state satisfies <<e>> P if it has an e-observation derivative $\Rightarrow$ e $\Rightarrow$ satisfying P. Using these new operators, we redefine the Can and Can't macros:

- <<Can e>> $\approx$ << e >> tt
- <<Can't e>> $\approx \sim$<<Can e>>.

EXAMPLE. Transition system D2 in Fig. 6 contains a hidden event "τ." D2 satisfies the following formulas:

- <<E0>> <<E2>> tt
- [[E0]] <<E3>> tt

The formula [[E0]] <<E2>> tt is not satisfied since after the E0-observation derivative E0.τ , we have <<E2>> ff.

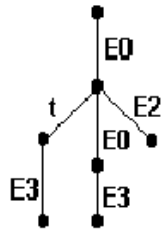

Fig. 6. LTS D2.

Table 10 shows μ-calculus definitions of abstract CSPE constraints.

TABLE 10
ABSTRACT CSPE CONSTRAINTS
IN THE PROPOSITIONAL MODAL μ-CALCULUS

Constraints	μ-calculus definition
~[[E1 ;→ E2]]	Eventually (<E1 > tt) ∧ Always ([[E1]] <<Can't E2>>)
a[[E1 ; → E2]]	Eventually (<E1 > tt) ∧ Always ([E1] <<Can E2>>)
p[[E1 ; → E2]]	Eventually (<E1 > tt) ∧ ¬(a[[E1 ; → E2]] ∨ ~[[E1 ;→ E2]])
a[[# ; → E1]]	<<Can E1>>

The following abstract CSPE constraints are true for labeled transition system D2 in Fig. 6:

- a[[E0 ;→ E3]]
- p[[E0 ; → E2]]
- ~[[E2 ; → E0]]

In [27], the following definition of a[[E1 ; → E2]] was used:

- Eventually (<E1>tt) ∧ Always ([[E1]] <<Can E2>>)

This definition may result in more p[[ ]] constraints.

The CSPE constraints in Tables 9 and 10 can be automatically checked against labeled transition systems using the Concurrency Workbench [42]. The Workbench is a software tool for the analysis of finite-state abstract programs written in CCS. The derivation of validity constraints from a labeled transition system D proceeds as follows:

1) CSPE constraints a[[E1 ;→ E2]] and ~[E1 ;→ E2] are defined in the modal μ-calculus using the macro facility of the Workbench. These macros can be "instantiated" for any pair of events (E1,E2). The CSPE macros are defined as in Tables 9 and 10 above.
2) For a user-specified pair (E1, E2), we instantiate a[[E1 ;→ E2]] and ~[[E1 ;→ E2]] and check which, if either, of these constraints holds in D. If neither constraint holds, then p[[E1 ;→ E2]] is generated in a post-processing step. (It is also possible to define and check p[[..]] within the Workbench. However, computing p[[..]] in a postprocessing step is simple and more efficient.)

For a user-specified list of observable events, we generate pairs of events that are then input into the Workbench for instantiation. The result of the Workbench analysis is a list of verified CSPE validity constraints for D.

ACKNOWLEDGMENTS

The authors acknowledge and thank Dan Duvarney and Jian Chen for their effort on the empirical studies reported in this paper. The authors gratefully acknowledge the anontmous referees for their helpful comments. Richard H. Carver's work was supported, in part, by the National Science Foundation under Grant No. CCR-9309043. The work of Kuo-Chung Tai was supported, in part, by the National Science Foundation under Grant No. CCR-9320992.

REFERENCES

[1] M.A. Vouk, D. McAllister, and K.C. Tai, "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-Tolerant Software," *Proc. ACM Software Testing Workshop*, pp. 74–81, July 1986.
[2] B. Beizer, *Software Testing Techniques*, second edition. Van Nostrand, 1990.
[3] R.N. Taylor, D.L. Levine, and C.D. Kelly, "Structural Testing of Concurrent Programs," *IEEE Trans. Software Eng.*, vol. 18, no. 3, pp. 206–215, Mar. 1992.
[4] R.D. Yang and C.G. Chung, "Path Analysis Testing of Concurrent Programs," *Information and Software Technology*, vol. 34. no. 1, pp. 43–56, Jan. 1992.
[5] S.K. Damodaran-Kamal and J.M. Francioni, "Nondeterminancy: Testing and Debugging in Message Passing Parallel Programs," *Proc. ACM/ONR Workshop Parallel and Distributed Debugging, ACM SIGPLAN Notices*, vol. 28, no. 12, pp. 118–128, Dec. 1993.
[6] G.H. Hwang, K.C. Tai, and T.L. Huang, "Reachability Testing: An Approach to Testing Concurrent Software," *Proc. Int'l J. Software Eng. and Knowledge Eng.*, vol. 5, no. 4, pp. 493–510, Dec. 1995.
[7] K.C. Tai, "Reachability Testing of Asynchronous Message-Passing Programs," Proc. *IEEE Int'l Workshop Software Eng. for Parallel and Distributed Systems*, pp. 50–61, May 1997.
[8] G.J. Holzman, *Design and Validation of Computer Protocols*. Englewood Cliffs, N.J.: Prentice Hall, 1991.
[9] G. v. Bochmann and A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing," *Proc. ACM Symp. Software Testing and Analysis*, pp. 109–124 1994.
[10] L.K. Dillon, G. Kutty, L.E. Moser, P.M. Melliar-Smith, and Y.S. Ramakrishna, "Graphical Specifications for Concurrent Software Systems," *Proc. 14th Int'l Conf. Software Eng.*, pp. 214–224, 1992.
[11] D. Helmbold and D. Luckham, "TSL: Task Sequencing Language," *Proc. Ada Int'l Conf.*, pp. 255–274, 1985.
[12] L.K. Dillon and Q. Yu, "Oracles for Checking Temporal Properties of Concurrent Systems," *Proc. Second ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 140–153, 1994.
[13] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," *IEEE Software*, vol. 2, no. 2, pp. 47–57, Mar. 1985.
[14] K.C. Tai and R.H. Carver, "Testing of Distributed Programs," ch. 33 of *Handbook of Parallel and Distributed Computing*, A. Zomaya, ed., pp. 955–978. McGraw-Hill, 1996.
[15] R.H. Carver and K.C. Tai, "Replay and Testing for Concurrent Programs," *IEEE Software*, pp. 66-74, Mar. 1991.
[16] K.C. Tai, R.H. Carver, and E.E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution," *IEEE Trans. Soft. Eng.*, vol. 17, no. 1, pp. 45–63, Jan. 1991.
[17] T.J. LeBlanc and J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. Computers*, vol. 36, no. 4, pp. 471–482, Apr. 1987.
[18] R.H. Carver and K.C. Tai, "Static Analysis of Concurrent Software for Deriving Synchronization Constraints," *Proc. IEEE Int'l Conf. Distributed Computing Systems*, pp. 544–551, May 1991.
[19] R.H. Carver, "Testing Abstract Distributed Programs and Their Implementations," *J. Systems and Software*, special issue on Software Eng. for Distributed Computing, pp. 223-237, June 1996.
[20] G. Bruns, *Distributed Systems Analysis with CCS*. Prentice Hall, 1997.

[21] K.C. Tai and R.H. Carver, "A Specification-Based Methodology for Testing Concurrent Programs," *Proc. Europe Software Engineering Conf., Lecture Notes in Computer Science 989*, W. Schafer and P. Botella, eds., pp. 154–172. Springer-Verlag, 1995.
[22] R.H. Carver and J. Chen, "Incremental Conformance Testing Using Lotos Specifications," *Proc. Fifth Int'l Conf. Computer Comm. and Networks*, pp. 42–47, Oct. 1996.
[23] P.V. Koppol and K.C. Tai, "An Incremental Approach to Structural Testing of Concurrent Software" *ACM Int'l Symp. Software Testing and Analysis*, pp. 14–23, Jan. 1996.
[24] A.J. Offutt, C.Z. Rothermel, and C. Zapf, "An Experimental Evaluation of Selective Mutation," *Proc. Int'l Conf. Software Eng.*, pp. 100–107, 1993.
[25] K.C. Tai, "A Graphical Representation of Rendezvous Sequences of Concurrent Ada Programs," *ACM Ada Letters*, vol. VI, no. 1, pp. 94–103, Jan./Feb. 1986.
[26] *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL*, K.J. Turner, ed., John Wiley & Sons, 1993.
[27] R.H. Carver and R. Durham, "Integrating Formal Methods and Testing for Concurrent Programs," *Proc. 10th IEEE Conf. Computer Assurance*, pp. 25–33, June 1995.
[28] K.C. Tai and Y.C. Young, "Synchronizable Test Sequences of Finite State Machines," 1998. to appear in *Computer Network and ISDN Systems*
[29] P. Tripathy and K. Naik, "Generation of Adaptive Tests from Nondeterministic Finite State Models," *Proc. Protocol Test Systems*, V, pp. 309-320, 1992.
[30] J. Arjjo, "On the Existence and Production of State Identification Machines for Labeled Transition Systems," *Proc. Formal Description Techniques*, VI, pp. 351–366, 1993.
[31] P.V. Koppol and K.C. Tai, "Conformance Testing of Protocols Specified as Labeled Transitions Systems," *Proc. IFIP Eighth Int'l Workshop on Protocol Test Systems*, A. Cavalli and S. Budkowski, eds., pp. 135-150, Chapman and Hall, 1996.
[32] G. v. Bochmann, R. Dssouli, and J. Zhao, "Trace Analysis for Conformance and Arbitration Testing," *IEEE Trans. Software. Eng.*, vol. 15, no. 11, pp. 1,347–1,356, Nov. 1989.
[33] Y.C. Young and K.C. Tai, "Observation Inaccuracy in Conformance Testing with Multiple Testers," *Proc. IEEE First Workshop on Application-Specific Software Eng. and Technology*, pp. 77-82, Mar. 1998.
[34] D. Rosenblum, "Specifying Concurrent Systems with TSL," *IEEE Software*, pp. 52–61, May 1991.
[35] E.J. Weyuker, T. Goradia, and A. Singh, "Automatically Generating Test Data from Boolean Specification," *IEEE Trans. Software Eng.*, vol. 20, no. 5, pp. 353–363, May 1994.
[36] A.M. Pardkar and K.C. Tai, "Test Generation for Boolean Expressions," *Proc. Int'l Symp. Software Reliability Eng.*, pp. 106-115, 1995.
[37] A.M. Paradkar, K.C. Tai, and M.A. Vouk, "Automatic Test Generation for Predicates," *IEEE Trans. Reliability*, pp. 515–530, Dec. 1996.
[38] K.C. Tai, "Theory of Fault-Based Predicate Testing for Computer Programs," *IEEE Trans. Software Eng.*, vol. 22, no. 8, pp. 552-562, Aug. 1996.
[39] P.C. Bates and J.C. Wileden, "High Level Debugging of Distributed Systems: The Behavior Abstraction Approach," *J. Software & Systems*, vol. 3, pp. 255–264, 1984.
[40] W.S. Cheng and V.E. Wallentine, "DEBL: A Knowledge-Based Language for Specifying and Debugging Distributed Programs," *Comm. ACM*, vol. 32, no. 9, pp. 1,079-1,084, Sept. 1989.
[41] C. Stirling, "An Introduction to Modal and Temporal Logics for CCS," *Lecture Notes in Computer Science 491*, pp. 1–20. Springer-Verlag, 1991.
[42] R. Cleaveland, J. Parrow, and B. Steffen, "The Concurrency Work-Bench: A Semantics Tool for the Verification of Concurrent Systems," *ACM Trans. Programming Languages and Systems*, vol. 15, no. 1, pp. 36–72, Jan. 1993,.

Richard H. Carver is an associate professor of computer science at George Mason University, Fairfax, Virginia. His current research interests include testing and debugging of concurrent software and computer-assisted learning.

Kuo-Chung Tai received his BS degree in electrical engineering from National Taiwan University in 1970 and his PhD degree in computer science from Cornell University in 1977. He is currently a Full Professor of Computer Science at North Carolina State University. From 1989–1991, he served as director of the Software Engineering program at the National Science Foundation. His current research interests include software testing, concurrent programming languages, and analysis, testing and debugging of concurrent software. He is an associate editor of *Computer Languages*, *International Journal of Software Engineering and Knowledge Engineering*, and *International Journal of Computer and Software Engineering*. He was co-program chair of the IEEE 1990 International Conference on Computer Languages and a co-program chair of the 1994 International Conference on Parallel Processing. Dr. Tai is a senior member of the IEEE, and a member of the IEEE Computer Society.