

# Projected State Machine Coverage for Software Testing

G. Friedman

Technion Israel Institute of  
Technology  
Technion City, 32000  
Haifa, ISRAEL  
+972-4-8405053

Qagalit

@techunix.technion.ac.il

A. Hartman, K. Nagin, & T.  
Shiran

IBM Haifa Research Laboratory  
Haifa University, Mt. Carmel 31905  
Haifa, ISRAEL  
+972-4-8296211

{hartman,nagin,shiran}

@il.ibm.com

## ABSTRACT

Our research deals with test generation for software based on finite state machine (FSM) models of the program specification. We describe a set of coverage criteria and testing constraints for use in the automatic generation of test suites. We also describe the algorithms used to generate test suites based on these coverage criteria, and the implementation of these algorithms as an extension of the Murφ model checker[4]. The coverage criteria are simple but powerful in that they generate test suites of high quality and moderate volume, without requiring the user to have a sophisticated grasp of the test generation technology. The testing constraints are used to combat the endemic problem of state explosion, typically encountered in FSM techniques. We illustrate our techniques on several well-known problems from the literature and describe two industrial trials, to demonstrate the validity of our claims.

## Categories and Subject Descriptors

D.2.5 [Testing and Debugging].

## General Terms

Verification, Validation

## Keywords

Automated test generation, finite state machine modeling, state machine projection.

## 1. INTRODUCTION

This paper addresses the problem of test generation from specifications of software systems. We assume that the system is specified in any one of the specification languages that enable an automated tool to derive an equivalent finite state machine (FSM) specification. Thus our techniques are equally applicable to specifications written in high level modeling languages such as

SDL, Statecharts, and Lotos, and to low-level specifications such as SMV, Murφ, or Promela.

We formulate the problem of test generation in an FSM as the selection of a finite set of execution sequences or paths, which satisfy a coverage criterion. A *coverage criterion* may be implicit, or may be expressed as an explicit list of *coverage tasks*. This formulation is both convenient and precise, however, the real goal of test generation is not to satisfy an abstract coverage criterion, but rather to uncover the faults in the implementation of the specification. These faults should be exposed using a finite budget for test generation and execution. The challenge is to generate test suites with effective fault detection properties, such that the larger the budget, the more effective the test suite.

There are several issues that limit the successful application of FSM models for the testing of concurrent programs. The first issue is that of state explosion. As the model of the software increases its fidelity to the program and its ability to predict the responses of the system to all possible stimuli, the number of states in the FSM grows exponentially. The second issue is that of test case explosion. The most commonly used coverage criteria for FSM models are state and transition coverage, which lead to an enormous number of test cases that usually cannot be executed within the testing time and budgetary constraints. A third issue in black-box testing of concurrent programs is the presence of non-determinism in the expected behavior of the program. By non-deterministic behavior, we mean that the specification allows for more than one acceptable response to a given sequence of stimuli. Moreover the responses to stimuli are not necessarily controllable by the tester. Carver and Tai [1] pointed out that there are certain classes of faults in concurrent programs that can never be detected by non-deterministic black-box testing. The problems associated with non-determinism are not addressed in this paper. A final issue limiting the usage of FSM based testing is that the skills required for defining an FSM model, and the tools for dealing with state and test case explosion are beyond the training and talents of the average software tester. We claim that the coverage criteria described here simplify the process of using these techniques, and we report on an industrial experiment showing that the costs of training can be compensated for by the increased efficiency of test generation.

The main contribution of this paper is the definition of a set of coverage criteria and testing constraints, which can be used for FSM test suite generation. We illustrate the uses of these criteria and constraints in combating the two inhibitors to FSM testing: the state explosion problem and the test case explosion problem.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA '02, July 22-24, 2002, Rome, Italy.

Copyright 2002 ACM X-XXXXX-XXX-X/XX/XXXX \$5.00.

We describe the test generation algorithms that can exploit these criteria in the context of an explicit FSM traversal tool. We also demonstrate, through a series of experiments, that the test suites based on these criteria are at least as effective at fault detection as the CSPE-1 constraints proposed by Carver and Tai [1]. (Briefly the CSPE-1 Constraints on Succeeding and Preceding Events are derived from the specification by considering all pairs of observable events and using the specification to decide under what conditions each ordered pair of events may occur. A coverage task for the CSPE-1 coverage criterion consists of an ordered pair of observable events and a verdict indicating whether this pair of events can be successfully executed in the current context. A test suite satisfying the CSPE-1 criterion contains sequences where each possible pair and each possible verdict occurs in some test case.)

We show how to apply our techniques to some standard problems from the literature and finally, we describe two successful industrial experiments that used these criteria and constraints to produce test suites with comparable defect detection to manually generated test suites, while saving on other testing resources.

## 2. PRELIMINARIES

A *finite state machine* (FSM) is a 6-tuple  $(S, I, A, R, \Delta, T)$ , where  $S$  is a finite set of states,  $I \subset S$ , is the set of initial states,  $A$  is the finite alphabet of input symbols, and  $R$  is the set of possible outputs or responses. The set  $\Delta \subset S \times A$  is the domain of the transition relation  $T$ , which is a function from  $\Delta$  to  $S \times R$ . The transition relation describes how the machine reacts to receiving input  $a \in A$  when in state  $s \in S$ , assuming that  $(s, a) \in \Delta$ . We interpret  $(s, a) \notin \Delta$  to mean that the input symbol cannot be received in that state. When  $T(s, a) = (s', r)$ , the system moves to the new state  $s'$  and outputs response  $r$ . If  $T$  is not a function, but rather a relation which associates each state-input pair with a non-empty set of state-response pairs, we say the FSM is *non-deterministic*, and we interpret the set as the set of possible responses to an input stimulus at a certain state.

In an FSM model of a software program, the state usually represents some aspect of the control flow of the program. For example, the state of an FSM model of a GUI may be the current screen on display; in a white-box FSM model, the state may represent the basic block of code currently being executed, or even the program counter register.

### 2.1 Extended Finite State Machines

An *extended finite state machine* adds a set of  $n$  variables  $x_1, x_2, \dots, x_n$  to each state representing the data fields used by the program in its execution. In fact, an extended finite state machine is just a finite state machine with additional information stored in each state, provided that the variables are all over finite domains. More formally, the state set  $S$  has the form  $S = D_0 \times D_1 \times \dots \times D_n$ , where  $D_0$  is the set of control states and  $D_i$ ,  $i = 1, \dots, n$  is the domain of the  $i$ -th state variable  $x_i$ .

Thus, an EFSM model of a program combines both control and data modeling.

We associate a labeled directed graph with a finite state machine  $(S, I, A, R, \Delta, T)$  as follows. The vertex set of the graph is the Cartesian product of the state set with the set of possible responses,  $S \times R$ . The labeled arcs of the graph are defined as follows. For each  $r \in R$ , there is an arc labeled  $a$  from the vertex  $(s, r)$  to the vertex  $(s', r')$  if and only if  $T(s, a) = (s', r')$ . Note that  $T(s, a)$  is undefined if  $(s, a) \notin \Delta$ . In the case of a non-deterministic EFSM, there may be more than one arc with the same label, leaving a given vertex of the digraph.

Note that when modeling several concurrent processes, the set of control states  $D_0$  may itself be the Cartesian product of the control state sets of each individual process.

### 2.2 Projected State Machine

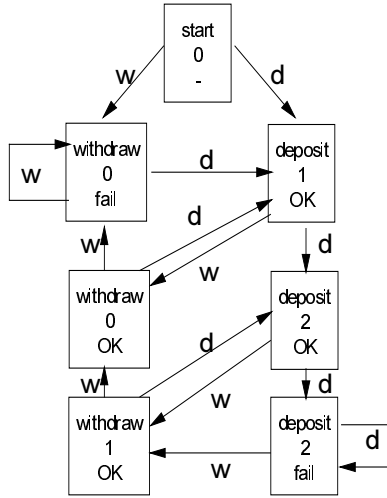
Let  $G = (V, E)$  be the arc-labeled digraph of an EFSM defined above. Let  $V$  have the form  $V = C_1 \times C_2 \times \dots \times C_m$  where each  $C_i$  is either a set of control states, a state variable domain, or an output result domain.

Given an equivalence relation  $\rho$  on  $V$ , we denote the equivalence class containing a vertex  $v \in V$  under  $\rho$  to be  $[v]$ . We define the *projected state machine graph* under  $\rho$  to be the labeled digraph  $G' = (V', E')$  where  $V'$  is the set of all equivalence classes under  $\rho$ , and there is an arc labeled  $a$  from  $[v]$  to  $[w]$  in  $E'$  if and only if there exists an arc labeled  $a$  from some member of  $[v]$  to some member of  $[w]$  in  $G$ . We will usually focus on the equivalence relation defined by the projection of  $V$  onto a Cartesian product of a subset of the sets  $C_i$ .

### 2.3 Two Place Buffer Example

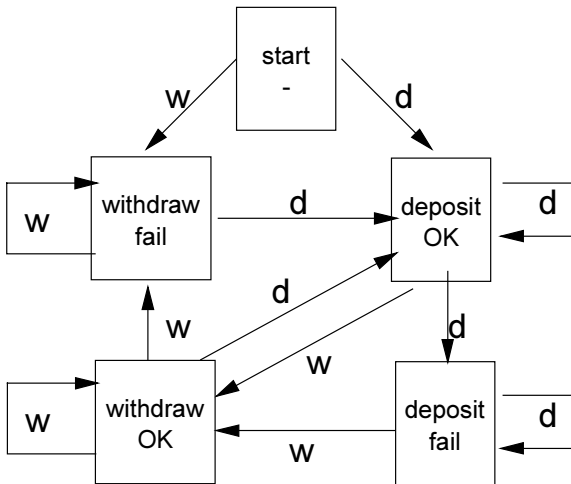
In Figure 1 we illustrate the labeled digraph associated with an extended finite state machine description of a two-place buffer. The state variables are **action**, which can take values from the domain {start, withdraw, deposit}, and **buffer** which can take values from the set {0,1,2}. The initial state is the ordered pair (start,0). The input alphabet is {d,w} – for deposit and withdraw respectively. The responses of the system come from the set {OK, fail}. A withdrawal succeeds whenever the buffer is not empty, and a deposit succeeds whenever the buffer is not full.

The vertices of the digraph are labeled by three values, the preceding action, the buffer occupancy, and the system response. The arcs are labeled by the input alphabet.



**Figure 1: Labeled digraph of the bounded buffer**

Figure 2 shows the projected state machine projecting onto the action and response coordinates.



**Figure 2: Projected state machine graph of the bounded buffer**

A test suite for the bounded buffer could consist of the following test cases:

TestCase 1: start, d(OK), d(OK), d(fail), w(OK), w(OK), w(fail).

TestCase 2: start, w(fail), d(OK), w(OK), d(OK).

Each test case is specified by a start command followed by a sequence of actions and the expected response in parentheses.

## 2.4 Test Suite Cost

In industrial settings, the testing of an application is constrained by the resources available. The aim of testing is thus to reduce the risk of defects being released to customers while not exceeding the time and material constraints available to the testers.

We model the cost of executing a test suite  $S$ , which contains  $n$  test cases with  $t_i$  transitions in the  $i$ -th test case as follows:

$$Cost(S) = A + nB + C \sum t_i$$

This cost function assumes that there is a fixed cost  $A$ , associated with carrying out the test, and that the set up cost  $B$ , for each test case, and the cost  $C$ , of executing each transition, is uniform for each test case and transition. The cost of the test suite for the bounded buffer given above is thus  $A + 2B + 10C$ .

The capacity of a test suite for finding bugs is much more difficult to measure, however one can compare the lists of coverage tasks used to create the test suite. If a list of coverage tasks properly contains another, then it is likely that the test suite generated for the larger set of tasks will, on average, detect at least as many bugs as the suite generated for the smaller set of tasks.

## 3. COVERAGE CRITERIA

Most of the literature concerning FSM test generation discusses the generation of test sequences, either on the basis of state and transition coverage, or on the basis of a set of test purposes crafted individually for each test case. The first approach often leads to an unacceptably large suite of test cases, while the second requires a large amount of work and expertise on the part of the test engineer. We propose a compromise between these two extremes, whereby the engineer focuses the test suite on particular state variables, output results, or control states, and combinations of these elements. Moreover, in the context of our graph, all these aspects are treated uniformly using the notion of a projected state machine.

Our test generation tool GOTCHA [2] provides a simple programming interface to specify coverage of various projections of the state graph of an EFSM. Note that a test case is a path in the full state graph, not in a projection, since paths in a projected state graph may not correspond to executable sequences of the software. We use the projected state graph as convenient shorthand for describing sets of coverage tasks.

### 3.1 Projected State Coverage

A projected state coverage criterion is specified in GOTCHA using the following syntax:

```
CC_State_Projection <boolexpr> ON <explist>
```

The <boolexpr> token can be replaced by any Boolean expression in the state variables, result fields, or control states, and the <explist> token can be replaced by any semi-colon separated list of expressions (not necessarily Boolean) in these same variables. The syntax that we use for expressions is the Murø syntax defined in [4]. It allows for most arithmetic and logical operators including quantification - but only over finite domains.

The projected state criterion asks the test generator to generate a test case containing a representative of each projected state in the EFSM projected onto the expressions in the list, provided the variables satisfy the Boolean expression. For example:

```
CC_State_Projection TRUE ON action; response
```

asks for a set of test cases, one passing through each of the five states in the projected state machine illustrated in Figure 2. The projected state coverage task (withdraw, OK) may be represented with buffer equal to 0 or 1, and this value will be chosen at random.

The following example illustrates the use of expressions to create more complex projections:

```
CC_State_Projection var1>0 ON var1; var2+var3;
```

asks for a set of test cases, one for each reachable instance of a projected state where `var1` is positive, and projecting onto the two values of `var1` and the sum of `var2` and `var3`. If each of `var1`, `var2`, and `var3` have domain  $\{0,1,2\}$ , there are ten potential coverage tasks, since `var1` can take the two values  $\{1,2\}$ , and `var2+var3` can take any value in the set  $\{0,1,2,3,4\}$ . The instantiation of the remaining variables is randomized, with equal probability of testing any reachable state in the equivalence class of the coverage task. The randomization algorithm is given in Section 5.

### 3.2 Projected Transition Coverage

A projected transition coverage criterion is specified in GOTCHA using the following syntax:

```
CC_Transition_Projection FROM <boolexpr> ON <exprlist> TO <boolexpr> ON <exprlist>
```

This allows the use of different projections and subsetting Boolean expressions for the source and target states in a transition. This criterion asks the test generator to generate a test case containing a representative of each projected transition in the EFSM projected onto the expressions in the list, provided the variables satisfy the Boolean expression.

For example, applying the criterion:

```
CC_Transition_Projection FROM TRUE ON action; TO TRUE ON action; response;
```

to the bounded buffer example, would produce a test suite with representatives of the eight projected transitions which correspond precisely to the CSPE-1 validity constraints with observable events `start`, `deposit`, and `withdraw` (see Table 1). The other ten possible projected transitions do not occur in the state machine. Furthermore, the transition chosen to represent (`withdraw` → `deposit`, `OK`) would be selected at random from among the three possible transitions in the full EFSM.

From	To
start	deposit, OK
start	withdraw, fail
deposit	withdraw, OK
deposit	deposit, OK
deposit	deposit, fail
withdraw	deposit, OK
withdraw	withdraw, OK
withdraw	withdraw, fail

**Table 1: Transition coverage tasks for the bounded buffer**

The test suite given in Section 2.3 covers each of the tasks described in Table 1. The notation used in the third column is Carver and Tai's.

## 4. TEST CONSTRAINTS

The most common drawback associated with EFSM modeling is the state explosion problem. The advocates of symbolic traversal

(e.g. [3]) and explicit traversal (e.g. [4]) make various claims, but both of these techniques suffer from worst-case behavior that requires exponentially large amounts of space. We do not claim to solve this problem, but we provide our test generator with tools to control the exploration of all reachable states. These tools require some expertise on the part of the user, but they do not require the user to learn the vocabulary of temporal logic since they only use declarative expressions.

The motivation for introducing test constraints into our test generator came from practical considerations associated with the translation and execution of test cases in a production environment. For example, we provide a means to specify the sets of states where a test case may end, thus facilitating the automation of a cleanup of the test environment before beginning the next test sequence. The implementation of the test constraints in the generation process enables their use in controlling the effects of state explosion as illustrated in Section 6.3.

### 4.1 Forbidden State Constraints

A forbidden state is specified in GOTCHA using the following syntax:

```
TC_Forbidden_State <boolexpr>
```

The `<boolexpr>` token can be replaced by any Boolean expression in the state variables, result fields, or control states.

The test generator interprets this constraint to mean that no state that satisfies the Boolean expression is allowed to appear in any test case it generates. The state is not stored in memory, and furthermore, there is no exploration of states that can be reached only from this state. For example, applying the following constraint:

```
TC_Forbidden_State buffer=2;
```

to the bounded buffer example, results in a state space containing only four states. Not only are the two states with `buffer=2` excluded, but the state (`withdraw`, `1`, `OK`) is also excluded, since it is only reachable from an explicitly forbidden state.

The original motivation for introducing forbidden states was to avoid generating test cases through parts of the specification that were either not yet implemented, or were known to be buggy. It also serves to divide up the state space under exploration, and enable the complete enumeration of a subset of the EFSM – and the generation of test cases in that subsection, thus providing a tool in the battle against state explosion.

### 4.2 Forbidden Path/Transition Constraints

The test constraint language also has syntax to specify forbidden transitions, forbidden paths, and other forbidden features of the state space. The forbidden transition is implemented as part of the EFSM traversal algorithm, and serves a similar purpose to the forbidden state syntax. The forbidden path is specified as follows:

```
TC_Forbidden_Path From <boolexpr> To <boolexpr> Length <intexpr>
```

This expression is interpreted as forbidding the generation of any test case that includes a subpath from a state where  $F$ , the first Boolean expression, is true to a state where  $S$ , the second Boolean expression, is true in  $k$  or fewer steps, where  $k$  is the value of the integer expression.

The constraint is implemented in the test generator by constructing a small FSM that tracks the number of steps since the last occurrence of  $F$ , and takes the product of this FSM and the EFSM under exploration. This implementation can either have the effect of ameliorating, or exaggerating, the state explosion problem. This is because the composite FSM has additional states due to tracking the constraint, but it also may have fewer states, whenever the constraint is violated.

We illustrate this phenomenon by noting that the following constraint on the two-place buffer:

TC\_Forbidden\_Path From buffer=0 To buffer=2 Length 3

reduces the number of states in the EFSM from 7 to 4, as in the forbidden state example. While introducing the constraint:

TC\_Forbidden\_Path From buffer=0 To buffer=3 Length 3

in the *three*-place buffer increases the number of states in that EFSM from 9 to 10. The extra state comes from the fact that the third of three consecutive deposits is a forbidden transition, whereas the state in which buffer=3 is reachable by a sequence of two deposits, a withdrawal and then two more deposits.

The forbidden path constraint is implemented by adding a new variable, say  $\mathbf{p}$ , to the EFSM, which computes the number of transitions traversed since the “from” condition was TRUE. The values assumed by  $\mathbf{p}$  are 0,1,...,  $k-1$ , and  $\infty$ , the latter indicating that  $k$  or more transitions have been traversed since the buffer was empty. The value of  $\mathbf{p}$  is set to 0 whenever the “from” condition holds, and is incremented whenever the new state does not satisfy the condition. Any transition from a state where  $\mathbf{p} \neq \infty$  to a state where the “to” condition holds is eliminated from the EFSM. This means that the number of states in the vicinity of states where the “from” condition holds increases, whereas states in the vicinity of states satisfying the “to” condition may be eliminated, or at least have their number of incoming transitions reduced.

Any other forbidden configuration that can be expressed as an FSM can also be implemented in a similar fashion. We have implemented other constraints of this type in GOTCHA. For example, we have implemented a constraint that requires all test cases that include two states satisfying some Boolean expression to be separated by a third state satisfying a different expression. In this way, we can ensure that a test case on a file system includes a data operation (read/write) in between any pair of file operations (open/close).

## 5. TEST GENERATION ALGORITHMS

The GOTCHA test generator accepts a behavioral model of the system under test, written in the Mur $\phi$  Description Language for EFSMs [4], together with a set of coverage criteria and test constraints written in the syntax given above.

The test generator builds a hash table containing all the reachable states in the EFSM using one of three graph traversal procedures. As the reachable set is built, the generator also builds a data structure that contains the details of all reachable coverage tasks.

Recall that a coverage task is either a projected state or a projected transition, which is an equivalence class of representative states or transitions in the full EFSM. The data stored for each coverage task include not only the projected data elements observed, but also the number of representatives of the task in the full EFSM,

and a pointer into the hash table pointing at a random representative of the task.

The user may chose between breadth first search (BFS), depth first search (DFS), and coverage first search (CFS). In each case, the generator keeps track of the states on the frontier of exploration; in BFS the frontier is a FIFO queue and in DFS it is a LIFO stack. In CFS, the states explored first, are those that lead to a previously unseen coverage task. In general, we found that breadth first search creates shorter test cases, but coverage first search decreases the overall length of the test suite, by inserting more coverage tasks per test case. The depth first search is useful in creating a single long test case when the overhead for setup and cleanup after a test case is prohibitive (large values of the cost parameter  $B$ ).

The algorithm for maintaining a pointer to a random representative of a coverage task is as follows. When the  $n$ -th representative of a task is discovered, replace the currently stored pointer with a pointer to the new representative with probability  $1/n$ , and retain the existing pointer with probability  $(n-1)/n$ . A straightforward induction argument shows that at the end of the state space exploration, each task has a pointer to a representative selected at random with uniform probability.

After completion of the state space exploration, a path is generated from an initial state to the representative of each coverage task, using the spanning tree created in the initial traversal. This path satisfies all forbidden configuration constraints, since forbidden configurations are eliminated from the search tree as soon as they are encountered. The only constraint that remains to be satisfied is the requirement to end the test case at a state satisfying some final condition. In most cases, this constraint can be satisfied by a breadth-first search of bounded depth starting from the task representative. In the worst case, no final state can be reached from the selected task representative, and another representative of the task is chosen. If no path to any representative of a task can be completed to a final state, the generator informs the user of the existence of a reachable but un-testable (under the test constraints) coverage task. Such tasks usually indicate a fault in the specification, as in the following example.

A reachable but un-testable task cannot be constructed for the  $n$ -place buffer since the digraph is strongly connected, but if we modify the model so that after a failed attempt to withdraw, the system does not allow the user to make a deposit – removing the arc from (withdraw, 0, fail) to (deposit, 1, OK), then the digraph is no longer strongly connected. In this modified EFSM, with the coverage criteria

CC\_State\_Projection TRUE ON action; response;

and the end test case constraint

TC\_End\_Test\_Case buffer=2;

the coverage task (withdraw, fail) has only one representative in the full EFSM. Moreover there is no path from this representative state to any state that satisfies the end test case condition.

The test generator also has an on-the-fly test generation algorithm, which can be used in the event of state explosion. When this algorithm is invoked, the generation of test cases is interleaved with further exploration of the state space. The advantage of this

procedure is that useful test cases can be generated even in very large state machines. The biggest drawback of on-the-fly test generation is that no coverage measure is available, since the number of reachable coverage tasks is only known at the end of the state space exploration. This drawback can sometimes be overcome, since an upper bound on the number of coverage tasks can be computed from the sizes of the domains of the projection variables or other ad hoc arguments. Another drawback associated with on-the-fly test generation is that one loses the guarantee of uniform randomization of the choice of representatives of coverage tasks.

## 6. THEORETICAL APPLICATIONS

This section describes several applications where our test generator was used, and compares the test suites we generated with those described in the literature. Carver and Tai [1] report results on the generation of tests for four applications: the bounded buffer, the readers and writers problem, the gas station problem, and the sliding window protocol. Müllerburg et al.[5] report on their experiments with the verification and validation of an elevator specification. We studied these five applications with our test generator and found that the use of our coverage criteria and test constraints simplifies the construction of test suites, without sacrificing the quality of the test cases produced.

### 6.1 Bounded Buffer

Carver and Tai describe a simple bounded buffer implementation involving the actions deposit and withdraw into a bounded buffer. The system responds with a success or failure, succeeding in a withdrawal if the buffer is not empty, and succeeding in a deposit if the buffer is not full.

Our model uses two state variables

- **buffer** – which counts the number of objects in the buffer and is initialized to zero.
- **action** – which records the last action performed, e.g., deposit, withdraw, or, create the buffer (start).

The CSPE-1 criterion used by Carver and Tai, with observable events *withdraw* and *deposit*, can be conveniently expressed using our syntax as:

CC\_Transition\_Projection FROM true ON **action**;

TO true ON **action**; **response**;

This coverage criterion guarantees that we will generate test cases that include the eight transition coverage tasks shown in Table 1. A test suite with 2 test cases and 10 transitions achieving this coverage criterion is given in Section 2.3.

We can generate a variety of other test suites by using larger or smaller sets of projection variables. For example:

CC\_State\_Projection true ON **action**; **buffer**; **response**;

generates a test suite with seven coverage tasks, one for each state in the entire EFSM (see Figure 1). The suite could have as few as 6 transitions (Start, d(OK), d(OK), d(fail), w(OK), w(OK), w(fail)). The corresponding transition projection generates 14 tasks, one for each arc in Figure 1. To cover these additional tasks one would need to add two test cases to the suite given above:

TestCase 3: start, d(OK), d(OK), w(OK), d(OK), d(fail), d(fail).

TestCase 4: start, w(fail), w(fail).

The following transition projection criterion generates one task for each of the twelve arcs in Figure 2.

CC\_Transition\_Projection

FROM true ON **action**; **response**;

TO true ON **action**; **response**;

### 6.2 Readers and Writers Problem

In the readers and writers problem, data is shared between several processes. Readers may have shared access to the data, but writers must have exclusive access. Carver and Tai describe seven different strategies to control how readers and writers gain access to the data. These seven strategies differ in the priority given to readers and writers who submit requests to read or write data. The seven strategies are subtly different, and they may each be regarded as a mutant or defective implementation of each of the other six specifications. The details of the seven strategies are given in [1].

We implemented all seven strategies in C++, and wrote a model of each strategy in the modeling language, both the models and the implementations were based on the specifications of each of the strategies. The models contained a number of data and control structures including: **queue** – a queue of the requests with a maximum length of 4 over the domain {Read, Write} and a **dataStatus** field over the domain { BeingRead, BeingWritten, FreeAfterRead, FreeAfterWrite }. For each strategy we generated a test suite using projected state coverage, projecting onto **queue** and **dataStatus**. Not all of the  $31 \times 4 = 124$  combinations of **queue** and **dataStatus** are reachable by the implementation; the test generator identified 92 reachable coverage tasks, and covered them with between 40 and 50 test cases of average length 15-20 transitions.

In our initial testing we uncovered eight minor defects in the C++ applications, and a defect in one of the models. Having debugged the models and applications, we then ran the test suites for each strategy model on each of the seven strategy implementations. In Table 2 below, we present the results of this cross-testing. The entries in the table are the first test case specified by the model  $M_i$  that detected a departure from specification  $i$  in the implementation of strategy  $j$ . Note that strategy 2 is included in the behavior of strategy 1, so the test suite for model  $M_1$  could not detect a fault in implementation  $I_2$ .

	I1	I2	I3	I4	I5	I6	I7
M1	-	-	2	6	2	5	2
M2	2	-	4	4	6	3	3
M3	1	5	-	12	6	4	4
M4	2	5	11	-	6	4	4
M5	5	5	5	5	-	42	5
M6	5	5	5	5	21	-	5
M7	1	3	3	3	6	3	-

**Table 2: Results of cross-testing readers and writers strategies**

Another set of test suites was also generated using state projection just on the **queue** variable. In this case, the test generator identified 31 coverage tasks that were covered in 15 to 20 test cases. These smaller test suites succeeded in distinguishing all strategies from each other, with one exception. The small test

suite for model M5 did not reveal any departure from its specification in the implementation I6. These results are consistent with Carver and Tai’s findings. Furthermore, the results are achieved without any analysis of the constraints and without requiring a deep understanding of the test generation techniques.

### 6.3 Gas Station Problem

We also modeled and implemented the gas station problem defined in Helmbold and Luckham [6] and created a test suite using Carver and Tai’s strategy with observable events (recorded in the state variable **action**) *pre-pay, activate pump, start pumping, end pumping, charge customer, and give change*. We record the response of the implementation in the state variable **response**. The CSPE-1 criterion can be expressed exactly as in the buffer example by:

```
CC_Transition_Projection FROM true ON action;  
TO true ON action; response;
```

The test suite generated will have the same fault detection properties as that derived by Carver and Tai, and by expanding or contracting the lists of projection variables we may obtain test suites with greater or lesser fault detection capabilities.

### 6.4 Sliding Window Protocol

The sliding window protocol (SWP) is an important protocol for the transfer of messages over unreliable channels. It requires the receiver and sender to exchange a positive handshake on each data transfer, and uses an acknowledgement window to support flow control.

The model we created for the SWP had four separate processes, the sender, receiver, and two buffers. The buffers store, forward, re-order, and lose both messages and acknowledgements. Our model follows the specification discussed by both Carver and Tai as well as those by Robin and Turner [7]. The messages are sent in a circular sequence over a finite range  $R$ , and the window size  $W$ , and buffer size  $B$ , are also parameters of the model. This parametric model enabled us to investigate the effectiveness of our test constraints in the event of state space explosion. We ran our test suites against the implementation of Robin and Turner, exposing a subtle defect in their implementation.

The main data fields of the model included variables that track the upper and lower edges of the window at the sender and receiver **SendWindow**, **RecWindow**, two Boolean arrays **SendTimer**, and **RecTimer** which record the state of the timers for each message and acknowledgement, and the buffers themselves **SendBuffer**, and **RecBuffer**.

Our model had 670000 states with  $(R, W, B)=(5,2,2)$ . When we investigated the case with  $(R, W, B)=(6,2,2)$  (the case investigated by Carver and Tai) our test generation engine was unable to complete the state space exploration using 50 MB of memory. To generate test suites in this case, we used two strategies: on-the-fly generation and state space partitioning using our test constraints.

The following pair of constraints partitioned the state space into two models with no overlapping coverage tasks related to the packet numbers.

```
TC_Forbidden_State "Large packet numbers forbidden"  
SendWindow.Lower >= 3;  
TC_Forbidden_State "Small packet numbers forbidden"
```

```
SendWindow.Lower < 3;
```

Using the following test constraint, we also determined that most of the state explosion was due to the number of possibilities of messages in the buffers. The test constraint was added to limit the contents of the buffers to contain only messages within the sliding window, or at most one message out of the window.

```
TC_Forbidden_State "No messages outside extended sender  
window in SendBuffer "
```

```
exists i:BufferNum_t do !isWithinWindow(SendBuffer[i],  
SendWindow.Lower-1, SendWindow.Upper)  
endexists;
```

The function **isWithinWindow(x,l,h)** returns true if packet number  $x$  is within the range from  $l$  to  $h$ , computing modulo the range  $R$ .

The implicit assumption here is that any packet in the buffer whose number is outside the window causes the software to behave in the same way as any other packet outside the window. So we choose to restrict our attention to a single packet number just outside the window, and forbid other packets from remaining in the buffer.

The above constraint, together with a similar restriction on the contents of the receiver’s buffer, reduced the size of the state space to 81655 states, enabling the generation of test suites with measurable coverage of this restricted behavior for a model with  $(R, W, B)=(6,3,3)$ .

Criterion	Varlist	Tasks	Test Cases	Transitions
CC1	rAction	6	5	111
CC2	sAction	12	9	216
CC3	s&rAction	30	16	377
CC4	rAction, rWindow	68	58	1788
CC5	sAction, sWindow	239	217	7483
CC6	s&rAction, rBuffer	810	581	19148
CC7	s&rAction, sBuffer	974	822	27132
CC8	s&rAction, s&rWindow	1421	1160	38836
CC9	s&rAction, s&rBuffer	4730	4114	137411
CC10	s&rAction, s&rWindow s&rBuffer	10541	9527	321470

**Table 3: Size of test suites for the sliding window protocol.**

We chose a hierarchy of coverage criteria and generated ten different test suites. Our model contains a variable **action** that keeps track of the stimulus to the system. To test the sender process, we set action to be from the set {start, sendData, resendData, receiveAck, hidden}, with all actions taken by the buffers or receiver labeled as hidden. To test the receiver process we used a different coverage criterion, hiding all the sender and buffer actions, and setting **action** to take values from the set

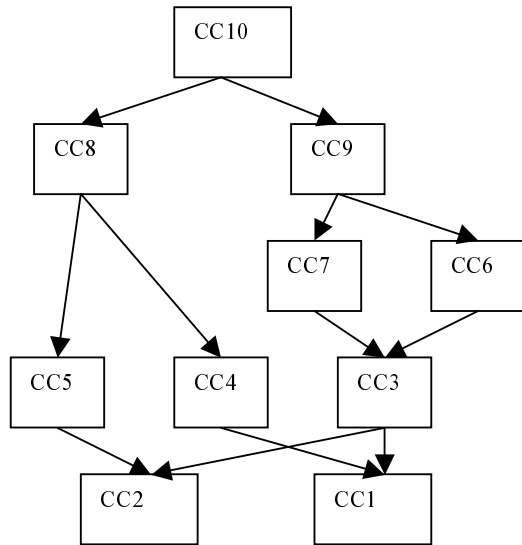
{start, sendAck, receiveData, hidden}. Each coverage criterion had the form:

```
CC_Transition Projection FROM action!=hidden ON Varlist;
TO action!=hidden ON Varlist;
```

In Table 3 below we list the coverage criteria, together with the sizes of the test suites.

The hierarchy of coverage criteria is illustrated below in Figure 3.

A criterion dominates another if all the coverage tasks in the second criterion are contained in the tasks of the dominating criterion.



**Figure 3: Hierarchy of sliding window protocol coverage criteria.**

The flexibility of our coverage criteria enables the generation of a number of test suites in a clear hierarchy, with increasing numbers of test cases, and numbers of transitions as the number of coverage tasks increases (see Table 3).

## 6.5 Elevator Control Model

Müllerburg et al.[5] report on the combination of systematic testing and formal verification of an elevator control program. Her group uses two separate specifications: one a behavioral model to generate test cases, and the other a logical, declarative model for formal verification.

Since the GOTCHA test generator is based on the traversal of the entire state space, it is easy to add a model checking capability for state or transition invariants. In fact, the same mechanism used for the forbidden state and forbidden transition test constraints performs model checking for these properties.

Müllerburg's experiment verified seven properties of the system, six of which can be expressed as state or transition invariants. Her test generation criteria can also be expressed easily in terms of state and transition projection criteria. Thus we were able to use a single model for both test generation and formal verification of invariant properties.

## 7. INDUSTRIAL EXPERIMENTS

In order to bridge the gap between theory and practice, we created a toolkit for creating test suites, translating the test suites to executable test scripts, and executing the test cases directly against the unit under test. The GOTCHA test generator forms the test generation part of the toolkit, and the remaining tools including test suite browser, test translation utility, and test execution engine are the TCBeans toolkit described in [2] and [8].

The testers who used the toolkit were given a four-day course in modeling and the use of the tools. The course was followed up with reviews of the models and test suites used in the industrial setting.

### 7.1 Distributed File System

The first industrial experiment repeated a test of a number of new features added to a file system shared by multiple hosts.

The records from the previous manually written test of these features were compared with the results of using the GOTCHA test generator and test execution framework.

The EFSM used in the test had 373248 states, and we generated 293 test cases to cover 729 coverage tasks. The test suite was generated using projected state coverage on a single data structure which records the actions performed by two processes concurrently accessing at most two of three directories. The observable actions are *open directory*, *read data*, and *close directory*.

In the original test, which utilized a total of 12 person-months, eighteen defects were found in the relevant features. The test suites generated by GOTCHA found 15 of these defects and found two new defects that had escaped the earlier test. The defects exposed included all 10 of the severity 2 defects found in the original test.

These results are significant since the personnel resources invested in the experiment – including training, model writing, test generation and execution – was 20% less than in the original test using traditional manual test generation.

The management of the testing unit involved was sufficiently impressed by the results of this pilot to commit to future use of the tools in subsequent releases of the product, and to recommend its use to other groups in the development laboratory. The methodology and tools have been applied to two other function tests since the original experiment.

### 7.2 Customer Support Center

A further industrial case study tested the dispatcher of an electronic customer support center. Customers of the support center contact agents and request support. The agents are represented by software components called access points. Each access point generates interaction objects to represent its communications. Interactions can include multiple access points. The observable actions correspond to access point methods. The model state is derived from the state of each access point, the data associated with each interaction, including the skills required of the agent to process the communication, and the state of the dispatcher queue.

The software includes a dispatcher that schedules interactions to be serviced by access points. The dispatcher has a complex

scheduling policy based on the skills required to deal with the communication, and the skills possessed by the available agents.

To create the test suite we used transition projection onto the access points and interactions data structures, while making sure that the test suite covered instances of an interaction being scheduled from the beginning, middle, and end of the queue. We also used a state projection criterion onto the access points to guarantee that the test suite covered interactions involving multiple access points. Two distinct models were written, one emphasizing the calls to the access point methods, and the other focusing on the dispatcher and its logic.

The first model had 77884 states, and the second had 42979 states. The suites exposed 37 bugs including one of severity 1, 21 of severity 2, and 14 of severity 3. An analysis of these defects showed that the majority of these bugs were exposed by the test cases with long sequences of calls to the API. Such test cases are difficult to construct manually, and exposed defects that would have escaped traditional test generation techniques.

The GOTCHA testing was done in parallel with other parts of the application being tested by more traditional means, and thus it is difficult to quantify the benefits gained from the projection-based testing. Many of the benefits were in the ease of adapting to changes in requirements, as the application matured. The use of a testing model enabled the adaptation of test suites to new requirements by making small changes to the model and re-generating the test suites. In the more traditional testing, such requirements changes created the need to edit each test script.

## 8. RELATED WORK

Much of the work on specification-based testing of concurrent programs has dealt with conformance testing of finite state machines. Several methods (see [9] and [10]) have been proposed for generating sets of test cases to verify that an implementation has the same sets of states and transitions as the specification. These methods concentrate on covering the states and transitions associated with the control of an FSM, and are more appropriate for protocol testing. These methods suffer from state explosion whenever any attempt is made to include data flow considerations into the FSM, or to express the system as a set of communicating FSMs. This state explosion also leads to an explosion of test cases, since the number of transitions to be covered also explodes.

Some methods that have been proposed for dealing with the state explosion problem are incremental testing and various reduction techniques [11], [12]. Our test constraints allow for both partition of the space under traversal, and also for the implementation of a primitive form of symmetry reduction, using a representative of a symmetry class, and forbidding states using other members of the class.

Many authors have created systems for the automatic generation of test cases. Both Hartmann et al. [13] and Offut and Abdurazik [14] generate test cases from UML specifications. The coverage criteria used in [13] was transition coverage of partitions of the data space. In [14] the authors propose coverage using a combination of explicit test purposes with transition coverage and predicate coverage of the predicates used in defining the transitions of the model. These, and most other forms of implicit test generation directives can be expressed in terms of the generic projected state and projected transition coverage we propose. The explicit test purposes used in the test generator of Jeron and Morel

[15] are more complex in that they specify a test purpose as a small FSM, but require a large amount of effort to create, and do not guarantee systematic coverage of the system under test. The coverage criteria introduced by Henniger and Ural [16] are comparable to ours in that they combine control and data dependencies, and they can be varied to create large and small test suites in a similar manner. Another approach to test generation criteria is taken by Amman et al. [17] who define a set of mutations of the specification and create a test case to distinguish each mutation from the correct implementation.

## 9. CONCLUSIONS

We described a method for specifying test generation controls for an EFSM specification of concurrent software. The controls include both coverage criteria, which instruct the test generator on what should be included in test cases, and testing constraints, which specify what should be avoided. The flexibility of the coverage criteria we propose enables a powerful mechanism for choosing test suites to suit the testing budget, without sacrificing efficiency or quality in the test generation process. Furthermore, our coverage criteria include Carver and Tai's CSPE-1 criteria as a special case. The testing constraints are a further weapon in the armory for fighting state explosion.

The algorithms we describe for test generation are applicable to any explicit traversal mechanism, and we have implemented them in a variant of the Murφ model checker.

Our test suites are successful in detecting faults both in synthetic examples taken from the literature and in an industrial setting.

## 10. ACKNOWLEDGMENTS

We would like to express our thanks to Richard Carver and Ken Turner for their helpful correspondence, and Ann Totten and Gary Bosko or their help in running the industrial experiments. We would also like to thank the anonymous referees of a previous version of this paper for their helpful suggestions.

## 11. REFERENCES

- [1] Carver, R.H., and Tai K.-C., Use of sequencing constraints for specification-based testing of concurrent programs. IEEE Transactions on Software Engineering, 24 (June 1998), 471-490.
- [2] Hartman, A., and Nagin, K.M., GOTCHA-TCBeans Tool Overview, Release 3.0.2, 2001  
<http://www.haifa.il.ibm.com/projects/verification/gtcb/documentation.html>.
- [3] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D., and Hwang, L. J., Symbolic model checking: 10<sup>20</sup> states and beyond, Information and Computation, 98 (1992), 142-170.
- [4] Dill, D., Murφ Description Language and Verifier, <http://sprout.stanford.edu/dill/murphi.html>.
- [5] Mullerburg, M., Holenderski, L., Maffei, O., Merceron, A., and Morley, M., Systematic testing and formal verification to validate reactive programs. Software Quality Journal, 4 (1995).
- [6] Helmbold D., and Luckham D., Debugging Ada tasking programs, IEEE Software 2 (1985) 47-57.

- [7] Robin, I. And Turner K. Protocol simulators, <http://www.cs.stir.ac.uk/~kjt/software/comms/jasper>.
- [8] Farchi, E., Hartman, A., and Pinter, S. S. Using a model-based test generator to test for standards conformance. IBM Systems Journal 41 (2002) 89-110.
- [9] Holzman, G.J., Design and Validation of Computer Protocols, Prentice-Hall 1991.
- [10] Bochman, G. v., and Petrenko, A., Protocol testing: Review of methods and relevance for software testing. Proceedings of ACM Symposium on Software Testing and Analysis (1994), 109-124.
- [11] Koppol, P.V., and Tai, K-C., An incremental approach to structural testing of concurrent software. Proceedings of ACM Symposium on Software Testing and Analysis (1996), 14-23.
- [12] Ip, C.N., and Dill, D. [Better verification through symmetry](#), Formal Methods in System Design, 9 (August 1996), 41-75.
- [13] Hartmann, J., Imoberdorf, C., and Meisinger, M., UML-based integration testing. Proceedings of ACM Symposium on Software Testing and Analysis (2000), 60-70.
- [14] Offut, J., and Abdurazik, A., Generating tests from UML specifications. International Conference on the Unified Modeling Language (1999).
- [15] Jeron, T., and Morel, P., Test Generation Derived from Model-checking, in Proceedings of CAV99, Trento Italy (Springer-Verlag LNCS 1633 1999), 108-122.
- [16] Henniger, O., and Ural, H., [Test generation based on control and data dependencies within multi-process SDL specifications](#). Proceedings of the 2<sup>nd</sup> Workshop of the SDL Forum Society on SDL and MSC (2000).
- [17] Amman, P.E., Black, P.E., and Majurski, W., Using model checking to generate tests from specifications. Proceedings 2<sup>nd</sup> IEEE International Conference on Formal Engineering Methods (December 1998).