Using a model-based test generator to test for standard conformance

by E. Farchi
A. Hartman
S. S. Pinter

In this paper we describe two experiments in the verification of software standard conformance. In our experiments, we use a model-based test generator to create a test suite for parts of the POSIX™ standard and another test suite for the specification of Java™ exception handling. We demonstrate that models derived from specifications produce better test suites than the suites specified by standards. In particular, our test suites achieved higher levels of code coverage with complete test requirements coverage. Moreover, the test suite for the Java study found code defects that were not exposed by other benchmark test suites. The effort involved in producing these models and test suites was comparable to the effort involved in developing a test suite by more conventional methods. We avoid the state space explosion problem by modeling only the external behavior of a specific feature of the standard, without modeling the details of any particular implementation.

In recent years, software modeling has enjoyed great popularity through the widespread adoption of object-oriented models as an aid to software design.[1] The use of software models for the generation of test suites has also been reported in both academic settings[2–5] and in practical experiments.[6–9] However, the specification-based modeling strategy for generating test suites has yet to reach widespread deployment in the software industry.

Software standards and language specifications are defined in natural language. Although they are usually supported by compliance test suites, each test suite is manually derived from the natural language description. As such, it is hard to determine if the compliance test suite is complete. Moreover, despite the huge investment of resources devoted to the preparation of such standards, they are still inherently ambiguous due to the use of natural language. We pose the following questions: Could a less ambiguous formal description be used instead of natural language? Can the conformance test suite that tests a standard implementation be derived automatically from the model specification?

The communications industry has used models written in SDL (Specification and Description Language), Estelle, PROMELA, UML (Unified Modeling Language), and others to investigate standards conformance.[10,11,3] These models are typically used for the verification of individual properties of the implementation and less frequently to generate conformance test suites. When they are used to generate test suites, several notions of coverage are applied.

In this paper, we show how certain complex aspects of software standards can be described using finite state machine (FSM) models. We investigate the POSIX** (Portable Operating System Interface) `fcntl` byte range locking feature and the Java** language exception handling feature as described in the standards. We also show that the test suites produced by these models are stronger than the conformance tests required by the standards. A larger case study is required to determine if the definition of commer-

cial software standards using formal languages is practical.

In our approach, we developed an FSM model to substitute for the natural language description. The test suite is automatically obtained from the model using an FSM model-based test generator named GOTCHA-TCBeans.[7] GOTCHA-TCBeans is built on top of the Murφ model checker.[12]

The FSM models can be reused to test different implementations of the specification. In addition, as the standard evolves, the model can be modified. As a result, the test suite evolves automatically with the standard. This innovation has the potential to change the entire process of standard development, implementation, and maintenance, which leads to improved quality of standard implementation. Finally, the GOTCHA model resembles a light implementation of the standard and is easily understood by developers. This approach to standard implementation enhances communication between testers and developers.

Previous work on test generation did not focus exclusively on the use of state machine models. Other techniques for test generation include reducing infinite domains to finite ones, data refinement, and syntactic coverage of the specification model.[13–15] Syntactic coverage of the specification model is analogous to code-based coverage of the implementation.[15,16] Our approach to coverage relates to the functionality and not to the syntax of the specification. Thus, the coverage criteria defined in our case studies are directly related to the semantics of the test objectives (e.g., test all types of lock collisions) and fit better with Marick's notion of test development.[17]

Automatic generation of test suites for protocol conformance is characterized by the existence of a formal specification for the protocol (typically in the SDL specification language[10]). Similarly, UML-based test generation assumes the existence of a UML specification that is used in the test generation process.[3] In contrast, we assume in our studies that software specifications use natural language, a situation typical for industrial software. Our studies indicate that the development of partial formal models focused on a specific feature or component of the software under test, for the sole purpose of test generation, is feasible and can provide good results in a realistic industrial setting.

We chose two very different case studies. The first study tested a subset of the POSIX[18] byte range locking standard API (application programming interface). The test cases are sequences of API invocations. In the second study, we tested the Java language exception handling feature. The test cases are programs in the Java language. In both case studies, the test cases are generated automatically from the FSM model.

The IEEE (Institute of Electrical and Electronics Engineers) POSIX Certification Authority and the National Institute of Technology Standards and Conformance Testing Group (SCTG)[19] offer a validation service and conformance test suites for POSIX. The use of the SCTG test suite is the accepted practice for testing conformance to the POSIX standard. We are not aware of any test suites produced for the POSIX standard that use FSM modeling.

In the second study, faults are hard to observe even when a defect occurs. This is due to the nature of the software under test. The component under test is a part of the garbage collector that analyzes the program control flow. This analysis is difficult when the program includes exception handling.[20,21] A serious fault in the garbage collector occurs whenever it collects live objects. If a defect occurs in the component (the map generator), the fault may not manifest itself. This depends on the specific program behavior. As a result, a combination of black box and white box testing strategies is required. Furthermore, complete code coverage is needed in order to fully exercise the map generator. Hence, the second study combines FSM modeling and automatic measurement of code-based coverage.

In the section "Strategies for software testing" that follows, we place our research within the state-of-the-art of software testing. Then, in the next two sections, "FSM modeling background" and "Modeling and test generation framework," we introduce the tools and techniques we used in our studies—FSM-based modeling and coverage-directed test generation. Next we describe our two case studies in "The POSIX byte range locking study" and "The Java exception handling study" sections. The last section, "Conclusions," contains our final comments.

Strategies for software testing

Two basic approaches to software testing are specification-based testing (black box) and program-based testing (white box). The black box approach

to testing[22,23] focuses on the externally observable behavior of a program under test, whereas white box testing utilizes internal knowledge of the program under test, such as the program control flow or the program data flow.[24] Research on software testing indicates that both approaches are useful in the effective detection of faults.[25] This paper focuses on specification-based, black box testing.

Two strategies for the generation of tests are risk analysis and coverage analysis. Under risk analysis, we include statistical-based testing and manual, risk-based, test generation. Under the heading of coverage analysis we include combinatorial design techniques and state machine enumeration techniques.

Risk analysis test generation. In statistics-based software testing,[26] a test suite is chosen using some probability distribution. The probability distribution attempts to capture usage patterns of the program under test. The probability distribution can be defined over the space of possible program inputs. Alternatively, a state of a program that accepts inputs is defined by the values of its internal variables. Each program state determines a different probability distribution over the set of program inputs, which yield a Markov chain. For example, if a file is opened for a read operation, a subsequent write operation is illegal, whereas a subsequent read operation is legal. In this case, the program state is determined by the way the file is opened. The history of the test results can be used to derive reliability measures, such as mean time to failure (MTTF) and test stopping criteria.

On the one hand, statistics-based software testing is good at producing formal estimates of program reliability. On the other hand, it is often hard to estimate the typical usage of the software and thus provide an accurate probability distribution.[26] Furthermore, the usage pattern of the software may change, thus requiring the software reliability measures to be re-estimated. Finally, if the cost of a fault is high, as in safety critical software, the testing process must guarantee that the software has no defects even under test scenarios that have very low probability in the usage model.

Strategies have been suggested in References 27 and 28 for the manual selection of test cases from a model in order to minimize the risk of defects escaping to the field. These strategies are applied to a UML model specification in Reference 27 and a version of a statechart model in Reference 28. These approaches are especially relevant as a supplement to automatic test generation.

Coverage-based test generation. Exhaustive black box testing of commercial software systems is impractical due to the size of the input space. Coverage techniques define a set of subdomains whose union contains the input space. The size of this set is usually much smaller than the size of the original input space. Only a representative of each subdomain is chosen in order to cover the input space. The choice of the subdomains is usually guided by some fault model. Test generation techniques have focused on coverage of the input space to meet the testing objectives.

Because the test designer specifies an entire test suite, instead of a single test, our approach requires less manual work than explicit test generation.

Combinatorial design strategies are used to obtain coverage. For example, the AETG** tool from Bellcore implements a test generation coverage technique based on combinatorial design.[29] In contrast to the state machine enumeration techniques discussed below, combinatorial-based testing techniques are scalable, but do not deal with the selection of stimuli sequences to test the implementation. Recent attempts to deal with this problem are reported for UML specifications using UCBT[30] (use case-based testing) and API testing using SALT.[31]

In state machine enumeration, different techniques such as state exploration have been used to generate the tests. All of these test generation techniques have worst-case exponential run time as a result of the reachability state space enumeration. Compared to the combinatorial design strategies, these techniques deal well with the selection of complex stimuli sequences. When using state machines, or any other automated test generation technique, the coverage criteria may be provided implicitly, by stating a general feature of the model, or explicitly by providing a list of all features to be covered.

Explicit test generation requires that the test designer choose what to test through the choice of an explicit test purpose (or goal) for each test case generated.

Automating the selection of individual stimuli in each test case requires user intervention and considerable manual input for each test case.[32] Test generation tools that support explicit test generation include: TGV, SAMSTAG, TVEDA, Verilog's ObjectGeode**, and Telelogic's Tau (see Reference 32 for references to these tools). All of these tools are SDL-based and have been applied primarily to telecommunication systems.

The test selection criteria used in implicit test generation are discussed by Horgan et al.[33] An example of implicit test generation is the extended message flow graph (EMFG) state exploration used in Reference 32 to automatically obtain a test suite. A data-flow-oriented criterion is used during state exploration to select the tests. This paper demonstrates the method with the all-use criterion. Others use coverage of the state machine structure as a test selection criterion. For example, ObjectGeode and Tau support structural coverage test generation of the specification.[34] Implicit test generation requires less manual effort compared to the explicit approach, but has less flexibility in the choice of test criteria.

In this paper, we present a different approach to test generation. A projection graph is derived from the specification state machine by the test designer based on some intuitive concern or fault model. During state exploration of the specification state machine, structural coverage of the projection graph is obtained. Thus, the test designer specifies an entire test suite instead of a single test as performed in explicit test generation. In view of the above, our approach requires less manual work than explicit test generation. In addition, the expressive power of the projection graph for specifying selection criteria is stronger than the selection criteria used in the implicit method and may be simpler and more natural than the explicit method. For example, given that the specification describes several processes and their interaction, a projection graph can easily be used to obtain the coverage of the transactions of a single process. This coverage requirement cannot be described by the implicit method and is not easily described by the explicit method, since a goal for each process transaction must be specified.

We apply our approach to the testing of the standard conformance of nontelecommunication software. We use an extension of the Murφ description language,[12] which is suitable for software modeling. Others have applied implicit and explicit state machine test generation methods to the testing of nontelecommunication software. The Unified Modeling Language (UML) is used for software specification. The UML statecharts were used in References 35 and 36 for implicit UML-based test generation. Prototypes UMLTest[3] and TnT[35] were integrated with the Rational Rose CASE (computer-aided software engineering) tool.[36] In Reference 37 the SCR (software cost reduction) requirement method was used to specify the software and derive implicit and explicit tests that structurally cover the state machine or verify that system properties are met. As explained in the previous paragraph, the use of state machine projection for test selection criteria differentiates our work from the explicit and implicit approaches.

FSM modeling background

Most software units can be viewed as reactive systems that receive stimuli from their environment and respond by emitting observable output signals and changing their internal state. A system is initialized in one of a known subset of states and its responses to stimuli depend only on its initial state and the sequence of stimuli it received. The system's behavior is specified in a specification document or a standard that describes the valid input stimuli in a given state and the set of acceptable responses to a given sequence of valid stimuli. As such, it is natural to model such systems by state machines. We provide an example of such a model in the following section.

FSM models of software behavior. A state machine is defined to be a 5-tuple (S, I, A, T, R) where S is a set called the *state set*, I is a subset of the states called the *initial state set*, A is a set called the *input alphabet*, R is a set called the *response alphabet*, and T is a subset of $S \times A \times S \times R$ called the *transition relation*. The state set and the input alphabet are finite sets. The interpretation given to the transition relation is that $(s, a, t, r) \in T$, if and only if the system, when in state s, reacts to input a by moving to state t and outputs response r. We say that input a is *valid* in state s if there exist t and r such that $(s, a, t, r) \in T$. We say that the software behaves *deterministically* if $(s, a, t, r) \in T$ and $(s, a, q, v) \in T$ imply $t = q$ and $r = v$.

When testing a software unit, it is important to validate the responses to a sequence of stimuli, as well as the internal state of the software after each stimulus is processed. Although this is not always possible, certain aspects of the internal state may be observable. It is common practice to view the state set S as a subset of the Cartesian product of sets D_1,

$D_2, \ldots, D_n$, where the D_i are referred to as the *domains* of the i-th *state variable* x_i, where some, but not all, of the x_i are observable. Our test generation tools require the domains and the input alphabet to be finite sets.

An *abstract test case* for a state machine consists of a sequence of input stimuli followed by the expected response and the values of the observable state variables expected to occur following the stimulus. An *abstract test suite* is a set of abstract test cases.

In this paper we represent a state machine by the labeled directed graph of the associated Mealy machine,[38] which is defined as follows: each node is labeled by an $(n + 1)$-tuple that contains the values of the state variables and an output response. Each arc is labeled by a member of the input alphabet. For each response r, a directed arc with label a connects from node (s, r) to the node (q, v), if and only if $(s, a, q, v) \in T$, (i.e., (s, a, q, v) is in the transition relation). An abstract test case is then just a directed path in this labeled directed graph whose initial node is an initial state.

Coverage criteria. The quality of a test suite for a software unit is often measured in terms of its coverage properties. The most commonly used coverage properties refer to aspects of the source code, such as statement coverage, branch coverage, or define-use coverage. For a catalog of software coverage models, see Reference 24. In the context of model-based testing, the most common coverage criteria are state coverage and transition coverage. For the models we use in testing standards compliance, the number of states and transitions is too large to make state or transition coverage a practical measure of the quality of any reasonably sized test suite. The coverage criteria we introduce in this paper are related to the coverage of the projection state machine model.

Let $G = (V, A)$ be the digraph of a state machine with node set V and arc set A. Further, let E be an equivalence relation on V, with $[v]$ denoting the equivalence class that contains v. We define the projection state machine graph $G[E] = (V[E], A[E])$ as follows: the nodes of the graph are the equivalence classes under E, so that $V[E] = \{[v] : v \in V\}$ and there is a directed arc from $[v]$ to $[w]$ if there exists an arc in G from some member of $[v]$ to some member of $[w]$. When the vertices are labeled by tuples, it is natural to consider the projection onto some subset of values in the tuple as the equivalence relation. For example, if the nodes are labeled by triples (x, y, z) we can project onto the first two variables using the equivalence relation defined by $(x_1, y_1, z_1) \sim (x_2, y_2, z_2)$, if and only if, $x_1 = x_2$ and $y_1 = y_2$.

A *coverage task* is an abstract concept that is either validated by an abstract test case or not, i.e., there exists a simple decision procedure to decide whether or not a particular task is validated by a particular test case. A test case that validates a coverage task is said to *cover* the task. A *coverage criterion* is a set of *coverage tasks*. An example of a coverage task is a node in the projection state machine graph. An abstract test case (path in the state machine) covers this task, if and only if a representative of the projected state lies on the path.

A natural criterion for the coverage of a standard is that every requirement of the standard is tested by some test case in the suite. We translate these requirements, in a natural way, to various projections of the state machine model. When we use *projection state coverage*, each coverage task is defined by an equivalence class of states. A test case covers such a task, if and only if it passes through a member of the equivalence class of states. We say that a member of the equivalence class is a *representative* of the task.

An abstract test case is always a path in the state machine graph. The paths in the state machine graph are always paths in the projection graph, but the converse is not true. It is possible to find a path in the projection graph that does not have a representative path in the state machine. Figure 1A presents a simple state machine with four states, $S = \{s, a, b, t\}$, and three transitions. The equivalence relation $E = \{\{a,b\}, \{s\}, \{t\}\}$ defines a projection graph (see Figure 1B). In the projection graph, there exists a path of length two from s to t, whereas in the state machine the only path from s to t has length three.

Test translation. Test cases produced from a state machine model are phrased in the abstract terms of the model. In order to generate executable test scripts these abstract test cases must be translated into concrete form. This involves creating a translation of the stimuli into execution statements and translating the expected responses and observable state variables into executable verification statements. The tools described in the following section include a framework for performing this translation.

Figure 1 (A) State machine graph; (B) projection state machine graph

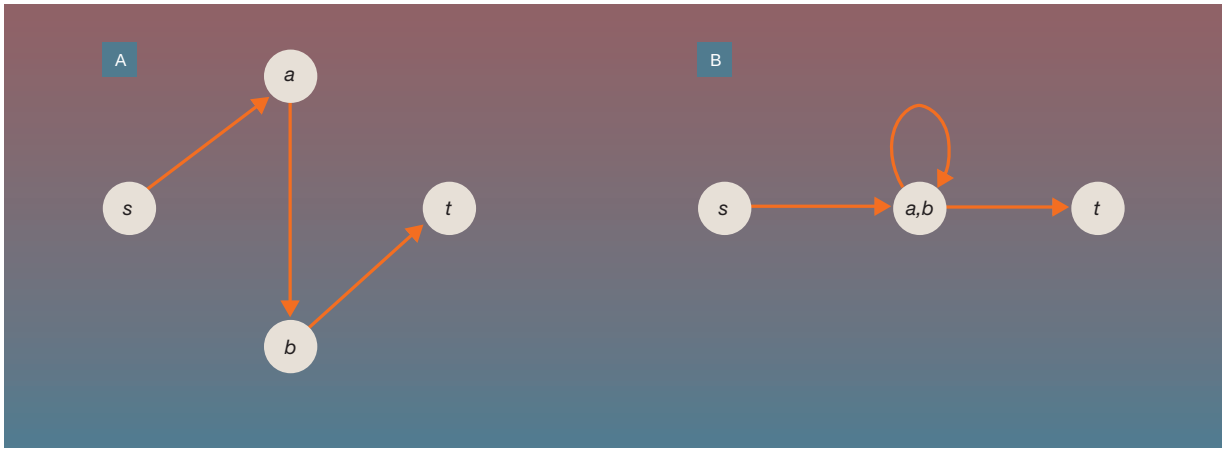
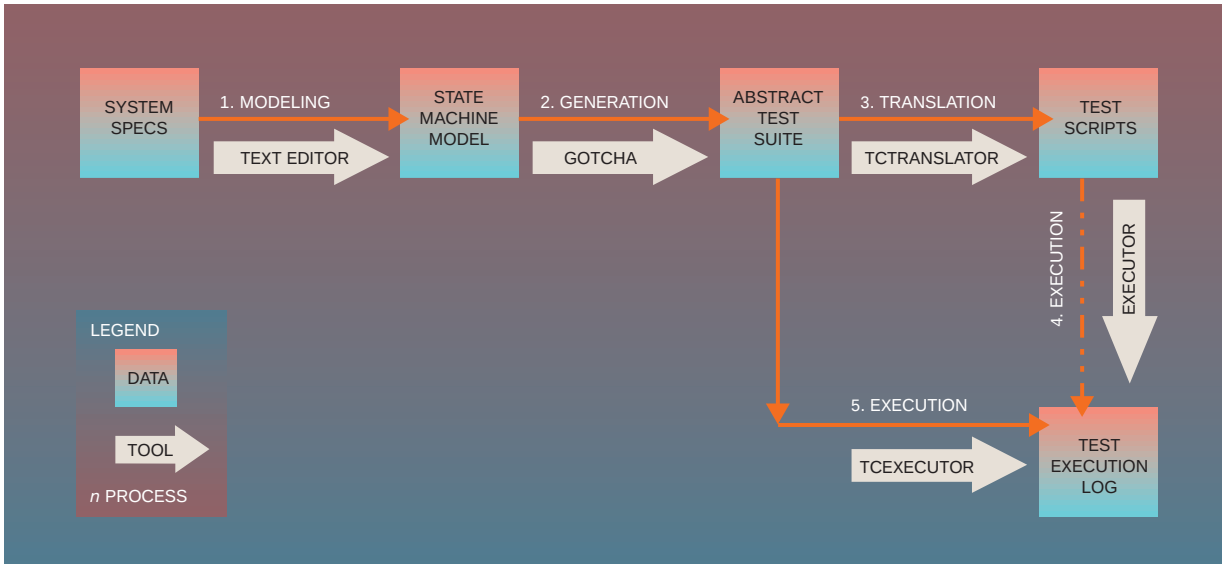


Figure 2 GOTCHA architecture



Modeling and test generation framework

In this section, we describe the GOTCHA-TCBeans modeling and test generation framework.

System architecture and methodology. We assume that the software under test is specified in some form, probably in a combination of natural language and diagrams or tables, that specifies the valid stimuli and the software's expected responses.

The first step in using our methodology to test the software is to create a state machine model of the specifications in the GOTCHA Definition Language (GDL). This language, described in further detail in the section "Modeling language," is a text-based language that extends the Murφ Description Language.[12] The user also uses GDL to write a set of coverage criteria and test constraints to direct the test generation. This is the process 1 (modeling) shown in Figure 2.

In process 2, the GOTCHA tool automatically generates an abstract test suite that satisfies the test constraints and covers each of the tasks specified by the coverage criteria.

The abstract test suite and a translation table written by the tester are the input to processes 3 and 5. The translation table for the TCTranslator tool can be written either in Java or in XML (Extensible Markup Language). A test execution engine executes this suite. In many practical situations the software under test has already been through a testing phase and a test execution framework already exists (process 4 in Figure 2). However, for a new product or one without an execution framework, TCBeans provides a tool, TCExecutor, that performs both translation and execution in a single step and creates a test execution log (see process 5 in Figure 2). The advantage of using TCExecutor is that the test log is in a format compatible with the abstract test suite and the faults detected are clearly mapped to the behavior that conflicts with the specifications.

Stack example. Here, we introduce the specification of a stack. The stack class public interface has five methods:

- `push(unsigned int i)`—pushes the element `i` onto the top of the stack. Returns `OK` if the stack is not full, otherwise returns the string `IMFULL`
- `pop( )`—returns the top element of the stack and `OK` if the stack is not empty, otherwise returns −1 and the string `IMEMPTY`
- `undo( )`—undoes the effect of the previous method call; returns `OK` if successful or otherwise the string `CANTREMEMBER`; two successive `undo( )` operations are considered unsuccessful.
- `delete( )`—destroys the stack and returns the memory allocated back to the system
- `stack(unsigned int i)`—creates a stack of size `i`. Returns `OK` if space successfully allocated

In the following example we specify a test suite that focuses on the three methods: `push(i)`, `pop( )`, and `undo( )`. We assume that the allocation of space and its return to the system are beyond the scope of the test plan.

Modeling language. GDL is used to describe the abstract notion of a state machine and projection graph. GDL consists of three parts: a section that declares the state variables and other global data, a section of procedures and functions, and a section that describes the valid stimuli and the software response to these stimuli. The syntax and semantics of GDL are taken from the Murφ description language (MDL), which is defined in Reference 12. In the following paragraphs we make the differences between GDL and MDL explicit.

The first section of the model description contains declarations of constants, types, and global variables. The global variables in the model are the state variables and result alphabet, which label the nodes of the state machine digraph. In the stack model we define:

- Constants for the maximum integer to be used in testing and the size of the stack to be tested:

```
Const MAXINT : 2;
Const STACKSIZE : 4;
```

- Type definitions for the return code, parameter for the push method, and result of the `pop( )` method. We have made a conscious modeling decision to create six return codes rather than using the `OK` code, for the successful response to `push( )`, `pop( )` and `undo( )`. This decision reflects our desire to know which method is called and to use this information in the coverage criteria specification.

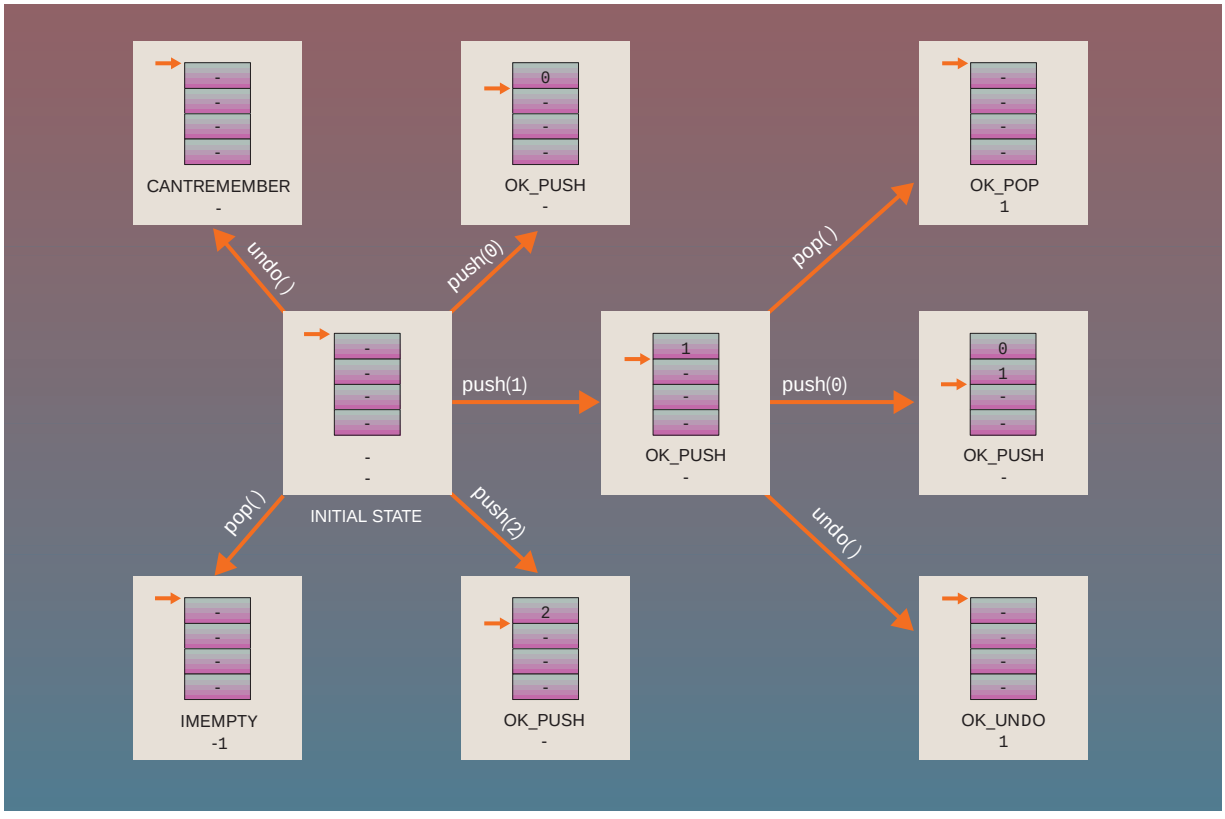
```
Type returnCode_t : enum {OK_UNDO, OK_PUSH,
   OK_POP, IMFULL, IMEMPTY, CANTREMEMBER};
Type unsignedInt_t : 0..MAXINT;
Type popResult_t : −1..MAXINT;
```

- Global state variables for the stack, its current size, and the output response:

```
Var Stack : array[1..STACKSIZE] of
     unsignedInt_t;
Var CurrentSize : 0..STACKSIZE;
Var Result : Record
code : returnCode_t;
popResult : popResult_t;
EndRecord;
```

The state variables define the labels on the nodes in the state machine graph. In this example, each node is labeled by a seven-tuple of values, (`Stack[1]`, `Stack[2]`, `Stack[3]`, `Stack[4]`, `CurrentSize`, `Result.code`, `Result.popResult`). Figure 3 shows a partial derivation of the stack machine state space. Each rectangle represents a possible state of the model state space. The stack is represented by a table

Figure 3 The state space of the stack model (partial view)



(`Stack[1,2,3,4]`) with an arrowhead that indicates its size (`CurrentSize`). The `Result` fields (`Result.code`, `Result.popResult`) are written below the table. An arrow between two rectangles is labeled by the stack operation that changes one state to the other.

In GDL, as in MDL, all the variables must be composed of either finite subranges of the integers or explicitly enumerated strings. This is to guarantee that the state machine specified is a finite structure that permits state enumeration.

The second section of the model contains declarations and descriptions of functions and procedures (possibly including local variables), used in succeeding sections of the model. These functions and procedures aid readability; they are not essential to the modeling process. For the stack model we define two procedures: `PushAction(i)` and `PopAction( )`, which encapsulate the behavior of pushing and popping, respectively, and enable their use by the `undo( )` method. In a real implementation of the stack class, these procedures are private class methods.

```
procedure PushAction(i : unsignedInt_t);
begin
    if CurrentSize = STACKSIZE
    then
        Result.code := IMFULL;
    else
        Result.code := OK_PUSH;
        for j := CurrentSize to 1 by -1 do
           Stack[j+1] := Stack[j];
        endfor;
        Stack[1] := i;
        CurrentSize := CurrentSize + 1;
    endif;
    clear Result.popResult;
end;
```

The clear macro sets every member of a data structure to its minimum value.

The procedure `PopAction( )` is similar. If the stack is empty, the procedure returns `IMEMPTY`; otherwise,

it sets `Result.popResult` to `Stack[1]` and updates the stack and its size accordingly.

The third section of the model contains valid stimuli specifications and the responses to these stimuli. These are transition rules that label the arcs (see Figure 3). Each transition rule is a command with a name, a precondition (a Boolean expression in the global and local variables), and an action (a block of statements that modifies the values of the variables). If the precondition evaluates to `TRUE` at a particular state, the method and its inputs are valid in that state. The action can be an arbitrarily complex statement block that contains loops, conditionals, procedure, and function calls. Sets of transition rules may be defined, where the rule's precondition and action receive one or more parameters. These `rulesets` are shorthand for writing a separate copy of the rule with each of the possible values of the input parameters. In the stack example we have a `ruleset` for the `push(i)` method and a rule for each of the methods `pop( )` and `undo( )` to be tested.

```
Rule pop( )
TRUE
==>
Begin
    PopAction( );
End;
Ruleset i:unsignedInt_t Do
    Rule "push(i)"
    TRUE
    ==>
    Begin
        PushAction(i);
    End;
EndRuleset;
Rule "undo( )"
TRUE
==>
Begin
    switch Result.code
    case OK_UNDO, IMFULL, IMEMPTY,
            CANTREMEMBER:
        Result.code := CANTREMEMBER;
        clear Result.popResult;
    case OK_PUSH:
        PopAction( );
        Result.code := OK_UNDO;
    case OK_POP:
        PushAction(Result.popResult);
        Result.code := OK_UNDO;
    endswitch;
End;
```

In this example, all the rules and all input parameters are valid in all states, so the precondition on each of the rules is simply `TRUE`. In more complex testing situations, it is often necessary to restrict the parameter values in order to avoid over-enthusiastic testing with illegal input values and impossible sequences of method calls.

Rules as defined above can only be used for deterministic models of software, since there is only one possible outcome state and result on the application of any given stimulus.

Coverage criteria and test constraints. The MDL has a syntax for describing the initial states of the state machine—it is simply a rule with no precondition. The initial state of the stack example is coded as follows (where `TC_StartTestCase` has been defined as a synonym for Murφ's `StartState`):

```
TC_StartTestCase "stack(STACKSIZE)- create
        Initial state"
Begin
    clear Result;
    clear Stack;
    clear CurrentSize;
End;
```

GDL defines other testing constraints, including a condition for ending a test case. In the stack example we use constraints to insist that each test case end with an empty stack as follows:

```
TC_EndTestCase "delete( )"
CurrentSize = 0;
```

This is a Boolean condition (not an assignment) that instructs the test generator that each test case should end at a state where this condition is `TRUE`.

GDL also contains a variety of test constraint syntax constructs other than the `TC_StartTestCase` and `TC_EndTestCase`. These include means for specifying forbidden states, forbidden subpaths, and other forbidden configurations to constrain the test cases generated. Since only `TC_StartTestCase` and `TC_EndTestCase` were used in modeling the standards described in this paper, we omit any detailed discussion of the other testing constraints.

The definition of four different coverage criteria in GDL is a major innovation that can be used to direct the test generator in its choice of test cases. These four coverage criteria direct the test generator to cre-

ate test cases according to the syntax and semantics described below.

Some state coverage is an explicit criterion that describes a single coverage task. The task representatives are states that satisfy a Boolean expression in the state variables. For example, specifying:

```
CC_Some_State "Failed Undo with full stack"
  CurrentSize = STACKSIZE & Result.code =
       CANTREMEMBER;
```

instructs the test generator to create a single test case passing through some state where the Boolean condition is true.

Some transition coverage is also an explicit test criterion that describes a single coverage task. The task representatives are transitions that pass from a state where the first Boolean expression is true to a state where the second Boolean expression is true. For example, specifying:

```
CC_Some_Transition "Empty the stack with an
          Undo command"
   From Result.code=OK_UNDO To Result.Code=
          IMEMPTY;
```

instructs the test generator to produce a test case with a transition from any state with `Result.code=OK_UNDO` to any other state with `Result.code=IMEMPTY`.

Projection state coverage describes a set of coverage tasks and not a single task. Each task is a state in a projection of the state machine. The projection variables are given in a list of expressions that follow the keyword `On`. A Boolean expression may also be used to further partition the projected state space. For example, specifying:

```
CC_State_Projection "Test that each value
       reaches the top of the stack"
CurrentSize > 0 On Stack[1];
```

instructs the test generator to generate a set of tests, one for each possible value of the variable `Stack[1]` and furthermore ensuring that the CurrentSize is strictly positive at each representative state. This is shorthand for three (0..MAXINT) separate coverage tasks. The first coverage task is the equivalence class of all states with `CurrentSize > 0` and `Stack[1]=0`, the second task is represented by any state with `CurrentSize > 0` and `Stack[1]=1`, etc.

The strength of the notation lies in its ability to specify a set of test cases, without knowing *a priori* how many tasks will be generated. For example:

```
CC_State_Projection "Check all possible
       result combinations"
    TRUE On Result;
```

instructs the test generator to generate a test case for each different value of the record `Result` observed in test. An upper bound for the number of possible `Result` records is 24, since there are 6 possible values of the code and 4 possible values of the `popResult`. Most of these combinations, however, are never encountered. A test case is generated for each projected state that is reachable from a `TC_StartTestCase` state. In the stack model, 11 of the 24 possibilities are observed (4 with `OK_UNDO`, 3 with `OK_POP`, and 1 with each of the other 4 return codes).

Projection transition coverage describes a set of coverage tasks. The syntax includes two Boolean expressions and two lists of expressions, one for the first state of the transition and one for the second state, so that:

```
CC_Transition_Projection "Check all
         transitions of the Result.code"
    From_Condition TRUE From Result.code;
    To_Condition TRUE To Result.code;
```

instructs the test generator to generate up to 36 test cases including a transition from each return code to each other. One such transition is the one from `OK_UNDO` to `IMEMPTY` specified above in the *some transition coverage* criterion. In the stack model, 24 of the 36 combinations are reachable.

Abstract test generation. The process of test generation is automated by GOTCHA, which explores the state space described by the GDL model. The user has several alternative test generation strategies, including breadth-first search, coverage-directed search, and on-the-fly test generation. Breadth-first search and on-the-fly test generation algorithms are well known. Coverage-first search involves giving priority to exploring states that lead to new coverage tasks before those that lead to areas of the projection state space that have already been encountered.

The principle that underlies GOTCHA's test generation strategy is the construction of a search tree that explores the entire state space. This is done by traversing all the reachable states of the state machine. The set of coverage tasks is constructed by observing each instance of a projection state or transition that satisfies a coverage criterion. An on-line randomization algorithm chooses a reachable representative of each coverage task encountered. GOTCHA performs a further reachability analysis, starting from the randomly chosen representative, to determine if a `TC_EndTestCase` condition can be reached from the specific instance of the coverage task. If a test that satisfies all the test constraints exists in the state machine graph, then the test case or path is output to a file in an XML format for describing paths.

If no `TC_EndTestCase` condition can be reached from a particular instance of a coverage task, another representative of the task is chosen. If no `TC_EndTestCase` condition is reachable from any reachable representative of a coverage task, then the user is notified of the fact that a reachable task has been identified with no test cases satisfying the test constraints through any of its representatives.

The abstract XML test suite comprises the following elements:

- The name of the model, for example,

```
modelname = "stack"
```

- A list of the state variables and their ranges, for example,

```
<StateVar name = "CurrentSize" range =
          "0..4" />
<StateVar name = "Result.code" range =
          "OK_UNDO, OK_PUSH, OK_POP,
          CANTREMEMBER, IMFULL, IMEMPTY"
          />
<StateVar name = "Stack[1]" range =
          "0..2" />
```

- A list of the rules or transition actions, for example,

```
<RuleDesc name = "push(unsignedint i)">
    <Param name = "i" range = "0..2" />
</RuleDesc>
<RuleDesc name = "pop( )" />
<RuleDesc name = "undo( )" />
```

- A set of test cases. Each test case consists of a sequence of rules, for example,

```
<Rule>
<MethodPattern> push(unsignedint i)
     </MethodPattern>
</DataInputPattern> i=1
     </DataInputPattern>
</Rule>
```

  followed by the state attained after the rule has been applied, for example,

```
<State>
CurrentSize = 1
Stack[1] = 1
. . .
Result.code = OK_PUSH
Result.popResult = −1
</State>
```

  The final element of a test case is a `TC_EndTestCase` rule name, for example,

```
<Rule>
<MethodPattern> delete( )</MethodPattern>
<DataInputPattern> </DataInputPattern>
</Rule>
```

Concrete test generation and execution. In both of the case studies reported here, TCTranslator is used to translate the abstract test suite into concrete test scripts. The translation table may be written in simple XML markup language or in Java. TCBeans creates a template for the translation table based on the abstract test suite so that the user is only required to fill in a few fields in the table in order to create the interface. Each element of the XML format in the abstract test case is given a translation template with simple substitution rules. For example, the `push(i)`, `i=2` rule in the stack example may be translated as `rc = MyStack.push(2);` every time it appears in an abstract test.

The State elements of the abstract test can be customized to compare `rc` with the abstract test variable `Result.code` and to output failure of the transition if they are not equal.

In general, the `Rules` are translated as stimuli to the system under test. The state elements are translated into verification statements to check that the response of the unit under test matches the expected

results predicted by the model. The `TC_StartTestCase` rules are translated into code that initializes the test case. The `TC_EndTestCase` rules are translated into clean-up code at the end of each test case. Using TCTranslator, the tester can supply a prologue and epilogue at the beginning and end of the test suite, respectively. In addition, if the state verification needs to be supplemented with additional checks, additional verification code at the end of each transition may be supplied.

A GOTCHA model for a subset of the POSIX standard helped generate a test suite of the interface that exceeds the standards testing requirements for compliance.

Both the abstract test suite and the suite execution trace can be conveniently viewed through the TCBeans browser. The browser displays the suite as a color-coded tree structure with panels for viewing the state variables and transitions in a test suite. The colors are used to indicate transitions that failed or succeeded during test execution.

The POSIX byte range locking study

The purpose of the Portable Operating System Interface (POSIX) standard is to define an operating system interface and environment based on the UNIX** operating system. This interface supports application portability at the C language source level.

The POSIX standard and its System Application Program Interface[18] are English-language documents. In this section we show that an aspect of the POSIX standard, the `fcntl` byte range locking APIs, can be described using a GOTCHA model. The model and its testing directives are constructed specifically to generate a test suite of the interface that exceeds all the standards compliance testing requirements.[19]

In 1999, the IBM Poughkeepsie Laboratory conducted a function test case generation study using GOTCHA. Certain parts of a POSIX-compliant subsystem[39] were tested again using GOTCHA; these parts included file I/O testing and stress testing. The resources used by the Poughkeepsie team for this testing pilot amounted to 10 person months, including the GOTCHA learning curve. This time is less than the time used to originally test the subsystem (12 person months). The test effort revealed two defects that led to documentation changes. In addition, postmortem analysis showed that 15 of the 18 defects found by the original function test effort would also have been found by this pilot test effort.[40]

The `fcntl` byte range locking test model described below is an expansion of one of the models used in the Poughkeepsie pilot. We have enlarged the model to ensure that all the POSIX standard compliance testing requirements are met.

Derivation of the byte range locking model. We begin by focusing on the data structures and their relationship to the standard.

When two or more processes are accessing a file, they can interfere with each other. The `fcntl` byte range locking interface provides control over open files so that interference between processes is regulated.

The standard states that a request for a shared lock should fail if the file was not opened with read access. We model the ways in which this file can be opened using the standard macros and an additional macro for modeling failure in opening a file.

```
/*
* O_RDONLY, O_RDWR, O_WRONLY are the
* standard macros used when a file is
* opened for read, read/write, and write
* respectively
*/
Type open_t: enum { O_RDONLY, O_RDWR,
     O_WRONLY, BAD_FILE_DESCRIPTOR};
```

The standard defines a file as a range of bytes. A lock is associated with a subrange of bytes in the file. There are two types of locks, shared locks (`F_RDLCK`) and exclusive locks (`F_WRLCK`). We chose to model the file as a two-dimensional array. The array is indexed by the byte offset in the file and the process accessing the file. The process also indexes an array that describes how the file was opened. A further Boolean flag is included in the file data structure to indicate whether or not the file was extended beyond its original size. In our model, we only allow a file to be extended once, although there is no such restriction in the POSIX standard. This restriction was introduced both to reduce the size of the state space and because defects that involve more than one extension of the file are beyond the scope of this model.

```
/* F_UNLCK, F_RDLCK and F_WRLCK are the
* POSIX standard macros for Unlocked,
* Shared, and Exclusive locks, respectively
*/
Type lockType: enum {F_UNLCK, F_RDLCK,
          F_WRLCK };
Type fileRange: 0..fileSize-1;
Type processRange: 1..numberOfProcess;
var file :
record
    open : array [processRange] of open_t;
    lockArray :
/* lockArray is a two dimensional array
* which reflects the lock type on every
* byte of the file held by every process.
*/
    record fileOffset :array[fileRange]
        of record processNumber
          :array[processRange] of lockType;
        end;
    end;
    extended : Boolean;
end;
```

These entities are the important aspects of the FSM state.

Having defined the data model, we then derive the FSM transitions from the standard. The model's transitions consist of operations on the file and operations on the processes. The file operations are the `fcntl( )` byte range locking operations and operations to open and close the file. The operations on the processes are signal, wake, and put to sleep.

The POSIX API standard states:

> When a shared lock has been set on a segment of a file, other processes shall be able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock shall fail if the file descriptor was not opened with read access.

The portion of the standard cited above is translated into a GOTCHA rule (or transition) that models the `fcntl( )` byte range operation. We assume that a transition is caused by calling the `fcntl( )` byte range lock command with parameters `l_type`, `l_start`, `l_whence`, and `l_len`. The names of these parameters are taken from the POSIX standard. The `l_type` parameter determines if the file was opened for a read operation, a write operation, or both. The `l_start` parameter determines the offset of the beginning of the byte range to be locked while the `l_len` parameter determines the length of the byte range to be locked. The `l_start` offset is calculated relative to the beginning of the file, the current file offset, or the end of the file. This is determined by the `l_whence` parameter.

```
if ((file.open = O_RDONLY) &
(l_type = F_WRLCK)) then
/* If the file is opened using O_RDONLY you
     cannot satisfy a write lock request */
return false;
endif;
. . .
/* Check if locking the lock segment is
                                    legal */
i := l_whence;
while ((l_start + i)
< (l_start + l_whence + l_len)) do
  j := l_start + i;
/*cannot set an exclusive lock if another
* process is holding a read lock
*/
if ((file.lockArray.fileOffset[j].
processNumber[otherPid] = F_RDLCK) &
(l_type = F_WRLCK)) then
/* Locking is forbidden: fcntl fails an
     error code is returned by a result
     variable*/
. . .
endif;
. . .
i := i + 1;
end;
```

In this model, state machine transitions are mapped to POSIX API invocations such as the `fcntl( )` API invocation described above. As a result, an abstract test represents a sequence of POSIX APIs invocations. In addition, as each transition rule models the return value of the corresponding API, GOTCHA generates the abstract tests in which the POSIX API invocation's expected result are provided. When an abstract test is run, TCBeans compares the expected result of each POSIX API invocation against the actual result of the API invocation of the system under test. In this particular case, TCTranslator is used to produce an input to an execution framework of the system under test that actually runs the test.

The semantics of a GOTCHA rule is that code describing a transition is executed atomically. This means that the model is much simpler than the actual im-

plementation that must deal with the concurrent process execution. We derive the rest of the model in a similar way.

Using coverage models to focus the abstract test suite. Five coverage criteria guide the test generator. Each of the criteria focuses on a different aspect of the standard. The three main criteria use state and transition projection to create sets of coverage tasks that include those required by the standard. Two further criteria ensure the coverage of two additional test requirements.

The first coverage criterion is directed at lock collisions. Two locks have a collision if their byte ranges overlap. The standard specifies when a shared lock or an exclusive lock can overlap. For example, the byte ranges of a shared lock can overlap the byte ranges of another shared lock, but cannot overlap the byte range of an exclusive lock. Our objective is to create tests that exercise all possible collisions. As a result, our coverage criterion requires that a shared lock and an exclusive lock be attempted whether or not there is an overlap with an existing lock. When there is an overlap with an existing lock, the overlap can occur either with a shared lock or with an exclusive lock. This coverage criterion includes the following six coverage tasks:

- (*shared, noOverlap*)
- (*exclusive, noOverlap*)
- (*shared, overlap, shared*)
- (*shared, overlap, exclusive*)
- (*exclusive, overlap, shared*)
- (*exclusive, overlap, exclusive*)

The coverage criterion is implemented by creating a special return code for each of these situations and specifying a coverage criterion that automatically guarantees that each return code occurs in some test. The GOTCHA code that specifies this coverage criterion is presented below.

```
CC_State_Projection /* A GOTCHA keyword */
ON
result: return_code_t;
```

GOTCHA generated tests that satisfy this criterion. As a result, 20 requirements of the POSIX test standard were met.

The second coverage criterion deals with waking up processes that wait on a lock. The requirements in the standard that deal with this case are modeled by a transition from `pid_state = ASLEEP` to `pid_state = AWAKE`. When such a transition occurs, the standard requires that three different collision types be observed. The result variable of the GOTCHA rule that models the `fcntl( )` operation tracks the occurrence of collisions of these types. Thus, we create a transition projection criterion, which projects the target state of the transition onto this variable as follows:

```
CC_Transition_Projection
FROM_CONDITION pid_state[1].state = ASLEEP
FROM TRUE;
TO_CONDITION pid_state[1].state = AWAKE
TO result: return_code_t;
```

The interpretation of this criterion is that a process transfers from a sleep (`ASLEEP`) state to a running `AWAKE` state while the values of the `return_code_t` are returned. The different values of the `return_code_t` represent different collision types.

This criterion resulted in the satisfaction of five additional requirements in the POSIX test standard.

Each of the standard testing requirements[19] is mapped to a coverage criterion, which ensures that the test generator creates a test suite that covers all these requirements. We covered 31 requirements using only five distinct coverage criteria. Our test suite covers additional situations not mentioned in the standard for measuring POSIX conformance. The test suite contained nine test cases, with an average of 15 API calls per test case.

There are five testing requirements in the standard that are not necessarily covered by our test suite. Three of these (D04, 51c, and 52d) are related to deadlock detection and are optional test requirements since they are implementation-dependent. The other two (33a and 50b) are related to lock inheritance when `fork( )` is invoked and to the system limit for total number of locks. To avoid state explosion and to improve readability, a different model should be written for the `fork( )` operation. It is our practice to avoid writing a complete model to meet one test requirement. As a result, these two test requirements are better covered using a hand-generated test.

The Java exception handling study

Java supports an exception handling facility that makes programs robust and simplifies the task of error handling. Error conditions are caught using the

try, catch, and finally constructs. However, the exception handling mechanism complicates control flow analysis and type analysis.

The Java Language Specification[41] describes the operational behavior of the exception handling mechanism and specifies which exception value should be returned in each case. The description includes 17 conditional statements nested up to four levels deep. This permits code with very complex execution flow, making flow analysis extremely complicated. In addition, the scopes of variables across exception boundaries complicate both data flow and type analysis. Efforts were made to formalize the exception handling specification for the purpose of verification.[42]

Control flow analysis of a program is useful for code-coverage analysis; for example, structural testing techniques,[43] regression testing (e.g., Reference 14), data flow testing (e.g., References 44, 45), program slicing, type analysis, precise garbage collection, program optimization, and verification. One such type analysis determines if the value of a variable or an operand stack entry has a reference or not.

In this case study, we tested the type map generator component of a type-accurate garbage collector.[46] For each Java method, this component generates the program control flow graph from the byte-code and analyzes the types of all the possible computed values (values of each local variable and each entry in the Java operand stack). For each possible garbage collection point in the code, the component generates a map that specifies the type of each variable and entry in the Java stack. The garbage collector uses the type information in the map to locate all the references on the program stack that point to objects on the heap at each execution point.

The most complicated part of this component is the generation and analysis of the exception handling control flow graph, sometimes referred to as the jsr problem.[20,21] The map generation algorithm is complex and the effect of a defective map on the program behavior is remote and indirect, thus making it hard to relate to the incorrect map. Some other prototype-based systems that use maps are the Jalapeño virtual machine[47] and Stichnoth et al.[48] Java garbage collector. There are no reports in the literature on the testing of these systems.

To tackle the complexity of the testing, we chose to model the exception handling feature of the Java language using GOTCHA, thus generating stronger test suites. The tests generated from the model are Java programs with complex combinations of the try, catch, and finally constructs. These programs exercise the exception handling mechanism and increase the code coverage of the component compared to the code coverage obtained by the standard JCK1.3 tests and the standard SpecJVM tests. In addition, these tests revealed four new defects in the component under test.

Java language exception handling feature. A Java method can throw two types of exceptions:

1. An unchecked exception is caused by a run-time error (e.g., division by zero) that can occur at any place in the method.
2. A checked exception is generated explicitly by a throw statement at a given point in the method.

A checked exception may be caught by some handler or can propagate to the calling method.

The try {block} catch {handler} finally {handler} statement catches an exception. The code within the try block is executed until either an exception is thrown or the end of the block is reached. If an exception is thrown, the catch clauses (when they exist) are examined to find the first clause that can handle the exception object. If the exception is not caught it will percolate back to the calling method. The rules that define when to propagate the exception object, what needs to be finalized before the propagation, and which environment to pass with the exception object (e.g., values of local variables) are very intricate.

The following code is a simple example of try statement use:

```
/* The handler class */
public class exceptionExample extends
    Exception{
    public exceptionExample ( ) {
    super( );
/* code for handling the exception */
    System.out.println("exception Handler");
    }
}
/* Example class */
public class Example{
. . .
public void M1exam( ) throws
    exceptionExample( ){
    . . .
```

```
    int value = someValue( );
    try{
...
     /* check condition for the exception */
     if (value == 0){
     /* throwing an exception */
     throw new exceptionExample ( );
     }
    . . .
    }
    catch(exceptionExample){
    /* do specific actions when the
                      exception is caught */
    . . .
    }
    finally{
    /* do specific actions when the last of
                        try or catch ends */
    . . .
    }
}
```

The try statement is implemented like a subroutine and is translated to byte code using the jsr (jump subroutine) and ret (return) opcodes. The jsr and ret opcodes provide the return address. Unlike subroutines, the return address points to the end of the try statement rather than the code that follows the place of the exception invocation. In addition, a new frame is not generated on the stack when the exception handler is invoked. These two features complicate control flow analysis and break the Gosling property.[49] The Gosling property is important for type analysis because it ensures that, if some instruction can be reached via multiple paths on which a local variable contains incompatible values (the type of values cannot be unified), then this local variable is unusable. The result of the Gosling property is the ability to obtain locally the type value of each variable and each entry on the operand stack for any code location.

Exception handling model derivation. We developed two GOTCHA models to test the correctness of the type analysis component. Both models were derived from the Java Language Specification.[41]

The first model corresponds to an FSM in which the legal transitions are explicitly given at each state and the transition rules follow the operational semantics provided by the language specification. The value of currentStmt determines the model state (see model description below). The currentStmt is used in the precondition of each rule that results in a model that mimics the language specification precisely. We illustrate the tight relation between the model and the language specification by providing parts of the try statement specification and its FSM model below:

- If the execution of a try block completes normally, then the finally block (if it exists) is executed and the choices (which are modeled by the finally state) are:

  The finally block completes normally . . .
  The finally block completes abruptly . . .

- If the execution of a try block completes abruptly because of a throw of a value V, then the choices are:

  There is a proper catch statement for V . . .
  Several options exist for the catch when it completes normally or abruptly (modeled by the catch state), with or without a finally statement . . .
  There is no proper catch for V . . .
  There are different options for the finally statement . . . (modeled by the finally state)

- If the execution of a try block completes abruptly for any other reason R, then the choice is:

  Different options for the finally statement . . . (modeled by the finally state)

The above standard specification fragment is used to create the segment of the GOTCHA model that produces try statements. The possible exception types are defined as an enumerated type (eit_t), which can take the values: no_exception, catchable, not_catchable, and system_exception.

```
/* set of transition rules
   (parametrized on exception_type)
   applicable within a try state */
Ruleset exception_type: eit_t
Do
Rule "create_try_block"
     currentStmt = try &
     (numCatches = 0 -> exception_type!=
                    catchable)
     ==>
     begin
        /* try block completes normally */
```

```
        if (exception_type = no_exception)
        then
           /* finally block is executed */
           if (numFinallys = 0)
           then
              currentStmt := end_test;
           else
              currentStmt := finally;
           endif;
        else
        /* the try statements completes
        abruptly throwing V,
        with a catchable exception,
        not_catchable exception, or
        system exception */
           if (exception_type = catchable)
           then
              currentStmt := catch;
           else /* system_exception or
                           not_catchable */
              if (numFinallys = 0)
              then
                 currentStmt := end_test;
              else
                 currentStmt := finally;
              endif;
           endif;
        endif;
        End;
End; /* end ruleset exception_type
                        (in a try block) */
```

GOTCHA generates an abstract test by firing in secession rules, such as the try rule described above. This translates directly to a Java program (see the subsection, "Abstract and concrete test cases," below).

The second model is based on the Backus-Naur Form (BNF) definition of the `try`, `catch`, and `finally` statements. We have five states: `try_s`, `catch_s`, `finally_s`, `end_block`, and `end_test`. A set of preconditions is associated with each state. Rather than specifying the rules to be followed at a given state, the preconditions are checked in order to find all the possible transition rules applicable. The test generator applies all the possible combinations when generating the tests.

The model has a set of transition rules for generating and throwing different exceptions (including unchecked exceptions), a rule for generating a `try` statement, a set of transition rules for generating `catch` statements, and a rule for generating a `finally` statement. Each rule may be used whenever its preconditions are met. A rule changes the current state of the FSM whose initial state is `try_s`. A test is generated by collecting a sequence of rules fired from the initial state and ending at the `end_test` state.

In the model fragment below we see that a `try_s` state can be entered if we are in any state other than the `end_block` state (used for closing a nesting level) and the `end_test` state. The `currentDepth`, `numTry`, and `numThrows` variables are used for controlling the type of tests that will be generated.

```
/* The try fragment of the BNF based model */
Rule "make_try( )"
(currentStmt[currentDepth] != end_block)
& (currentStmt[currentDepth] != end_test)
  & (currentDepth < MAX_DEPTH)
  & (numTry < MAX_TRY)
  & (numThrows[currentDepth] = 0) /* do not
      generate a try after a throw (>= 1)*/
    ==>
    Begin
        numThrows[currentDepth] := 0;
        currentDepth := currentDepth + 1;
        currentStmt[currentDepth] := try_s;
        numTry := numTry + 1;
    End;
```

GOTCHA generates all possible sequences of states and rules (abstract tests), which the TCTranslator translates to the corresponding Java programs (see the subsection "Abstract and concrete test cases," below).

The resulting model differs from the first model in that the body of the transition rule does not contain a control flow. Instead, preconditions are used to restrict the firing of transitions in the model. This model generates far more interesting test cases, but seems more remote from the original standard definition. One of the reasons this model generates better test cases is that its structure enables deeper levels of recursion in the test cases before state explosion is encountered. The state explosion problem is mitigated in this model by the use of seven testing constraints that forbid certain kinds of test cases and, thus, limit the reachable state space. For example, we limited the generation of exceptions in the try clause at the highest level of nesting using the test constraint statement given below. The statement forbids the generation of a `system_exception` whenever a `try` statement occurs at nesting level 0 in the Java program generated as a test case.

```
TC_Forbidden_State ExceptionType[0][try_s] =
    system_exception;
```

Abstract and concrete test cases. GOTCHA generates abstract tests from the model by giving concrete values to the state variables and creating sequences of rules that are later interpreted as the statements in a Java program that uses the `try`, `catch`, and `finally` constructs.

An abstract test consists of a sequence of rule invocations and the resulting state entered. A sample abstract test and the resulting test (Java program) are given below:

- `init( )`—generates a new method (`test`) and opens the outer try block
- `make_try( )`—opens an inner `try` block
- `make_catch(Exception1)`—closes the inner try block and inserts a catch clause for the inner `try` block catching `Exception1`
- `make_sys_throw( )`—throws a system exception inside this `catch` clause
- `make_end( )`—closes the `catch` clause
- `make_finally( )`—closes the outer `try` and inserts a `finally` clause for the outer `try` block
- `make_throw(Exception1)`—throws `Exception1` inside the `finally` clause
- `make_end( )`—closes the finally clause
- `make_done( )`—ends the test case method

Each transition rule is responsible for generating part of the test case. For example, both `init( )` and `make_try( )` generate a `try` statement and open a `try` block.

The abstract test is then translated by the TCTranslator to a Java test program. The Java code generated for the above abstract test is:

```
public static void TRY121( ) throws
   Exception{
try{
System.out.println("TRY121( ): Try
   statement");
if(false) throw new Exception1( );
try{
System.out.println("TRY121( ): Try
   statement");
if(false) throw new Exception1( );
}
catch(Exception1 e){
System.out.println("TRY121( ): catching
   Exception1");
System.out.println("TRY121( ): system
   exception ");
int a=1, b=0, c=a/b;
a=c+5;
}
}
finally{
System.out.println("TRY121( ): Finally
   statement");
System.out.println("TRY121( ): Throwing
   exception Exception1");
if(true) throw new Exception1( );
}
}//end of TRY121( )
```

TCTranslator uses XML markup language to describe the translation from abstract test case elements to concrete test case elements. The `<RuleDesc>` tag describes the text to be substituted in the concrete test case for each transition rule in the abstract test case. When the transition rule `init( )` occurs in an abstract test, it is translated into the header and the first `try` statement of the Java program, which is the concrete test case. The XML code for this translation is presented below (`_a2c_#test` is a run-time variable that is replaced by the abstract test case identifier):

```
<RuleDesc desc="init( )">
public static void TRY_a2c_#test( ) throws
   Exception{
try{
System.out.println("TRY_a2c_#test( ): Try
   statement");
if(false) throw new Exception1( );
</RuleDesc>
```

The TCBeans markup language also contains clauses for generating validation code in the concrete test suite, using the abstract test as an oracle for comparison with the output of the application under test. The validation tags were not used in this case study.

Verifying the test results. In this study, we do not use the model for the automated generation of expected test results. Instead we built a system that generates type information at run time and checks the correctness of the corresponding type maps. This check is necessary since faulty type maps may not cause a program failure. Erroneous maps may identify a slot on the stack as a reference when it is not or miss a reference, which may cause the collection of a live object. In the second case the program may not fail if there is another reference to the object or

if the garbage collection starts after the last use of the object.

The above system was implemented by instrumenting the JVM (Java virtual machine) interpreter to perform direct verification of the data contained in each type map. The instrumentation generates type information for operands and local variables during run time. To minimize the changes to the original JVM code we stored this information on a "shadow" stack manipulated in parallel to the JVM interpreter stack. Upon entry to a method, the information of the shadow stack is initialized with the types of the method arguments. The type information on the shadow stack is updated following the execution of each bytecode that affects the stack.

Whenever the execution of a method reaches a point for which a type map exists (the maps are generated when the method is first loaded), the map is retrieved from the repository and compared with the type information on the shadow stack. The instrumented code reports an error whenever there is a mismatch between the information on the shadow stack and the type map values.

Experimental results and the strength of the exception handling model. We used the test suite generated by our model as a stress test for the type map generator component of a new type-accurate garbage collector.[46] The map generator package contains four files, written in C, which include 6914 lines of code distributed over 2681 basic blocks.

We used ATAC[50] to measure statement coverage. Each defect was repaired before testing continued. We first tested the map generator component by running the JCK 1.3 (Java Compatibility Kit) compliance and SpecJVM tests. These test suites exposed 16 defects (3 design defects and 13 implementation defects) and reached 78 percent statement coverage. We then ran all test cases generated by our second abstract model. Four new defects (2 design and 2 implementation defects) in the corrected map generation code were found and the statement coverage rose to 83 percent. Our final testing activities involved manually creating specific test cases aimed at a level of abstraction lower than that exposed in the model (bytecode, rather than Java source code). These hand-generated test cases exposed one further design defect.

The defects found using the automatically generated test suite are characterized by the complex nature of the jumps and returns from the blocks of code generated from a `try catch finally` combination. A representative defect of this type occurred when two blocks of code, `A` and `B`, jumped to the same `catch` block. The local variable `x` is a reference in `A` and it occurs in `B`, but not as a reference. The variable `x` is not referenced in the `catch` block, so the `catch` block could be consistently executed. However, at the exit from the `catch` block, the type of `x` had to be retrieved from a map dependent on whether the `catch` block was entered from `A` or `B`. To perform this correctly, the map generator should have generated and stored two distinct maps, but in the defective code only one map was generated and stored.

Conclusion

In this paper we demonstrated that behavioral models can be derived from software specifications and used for the creation of test suites for standard compliance. In the first study, the automatically generated test suite covered the POSIX standard testing requirements.[19] In addition, we demonstrated in each case study that the test suite is of high quality by running it against an implementation of the specification and locating defects that were not found by more traditional test suites. We also demonstrated in the Java exception handling case study that the test suite considerably improves the code coverage attained by more traditionally generated test suites.

In each study the effort involved in creating the test suite was surprisingly low. The human resources required for the first study, including the one-time investment of learning new tools and methodologies, was 17 percent less than the time spent testing the file system by conventional methods. In future use of the techniques, one could reasonably expect additional resource savings. In the second case study, the time spent on modeling and testing was three to four person months, approximately half the time spent on similar systems (see, for example, Somerville's book[51]).

The skills required for the use of our techniques are not usually found in existing testing organizations. Testers with some programming knowledge can acquire these skills quickly. In the first case study, the testers received an intensive four-day course on the tools and methodology; in the second study, the testers studied the manuals and had a two-day course.

A well-known drawback of FSM modeling is the tendency for models to suffer from a combinatorial ex-

plosion. We have two strategies for dealing with this problem. Our modeling language has testing constraints that enable the state exploration algorithm to prune the search tree. We also use on-the-fly test generation to create test cases in the exploration of extremely large state spaces. The drawback of the on-the-fly approach is the loss of a functional coverage guarantee. On-the-fly generation was not necessary in either of the case studies.

A further issue with automatic test generation is the tendency of test suites to grow out of control. We treated this issue by using coverage criteria to shape and limit the size of the test suite. Our projection state and projection transition coverage criteria can be used to create a focused test suite with the appropriate randomization of nonessential variables that increase the likelihood of fault discovery.

A more general conclusion concerns the applicability of our results to the creation and maintenance of software standards. We have shown that a model of some specification of a standard can be efficiently used to both specify the standard and later generate conformance tests of high quality. This leads us to recommend that the standards themselves be written as a set of formal models. This increases the maintainability and accuracy of the specification and enables both formal verification and automatic test generation as well as possibly performance modeling and simulation experiments. This approach is already in use in the telecommunications industry, where there are ITU (International Telecommunications Union) standards for modeling languages (e.g., SDL) and test suites (e.g., TTCN, or Testing and Test Control Notation).

A possible objection to this approach is the difficulty of handling a massive standard such as POSIX in a single finite state machine model that captures all of its interesting external behavior. The POSIX standard consists of a series of component-level functionalities, each of which can be modeled separately with limited interactions between the components. The standard's test requirements are also decomposed along these lines. For example, it is possible to separate the process primitives from the file and directory operations. This makes a component-by-component modeling approach feasible.

A further conclusion relates to the use of these techniques in general software development processes and not just in the testing of standards conformance. When the software has a well-defined specification document and the quality demands of the product justify a significant testing effort, then these methods have proved their efficiency in the case studies. In both case studies, we found it necessary to augment the automatically generated test suites with a few well-chosen manual tests. The use of model-based strategy and automatic test generation is not a silver bullet to solve all testing problems; however, the evidence presented here indicates that it is a valuable weapon in the developer's armory and has clear advantages over the more traditional approaches.

Acknowledgments

The authors would like to thank the GOTCHA TCBeans team: Kenneth Nagin, Andrei Kirshin, and Sergey Olvovsky for helping us use their system; Niv Buchbinder for helping with the testing of the Java case study; and Ann T. Totten from the Poughkeepsie IBM laboratory for running the tests of the first study.

Cited references and notes

1. G. Booch, *Object Oriented Analysis and Design with Applications*, *2nd edition*, Benjamin Cummings, San Francisco, CA (1994).
2. J. Callahan, F. Schneider, and S. Easterbrook, "Automated Software Testing Using Model-Checking," *Proceedings of the 1996 SPIN Workshop*, Rutgers University, New Brunswick, NJ (1996), pp. 118–127.
3. J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," *Second International Conference on the Unified Modeling Language (UML99)*, Springer-Verlag, New York (1999).
4. A. Paradkar, "SALT: An Integrated Environment to Automate Generation of Function Tests for APIs," *Proceedings of the International Symposium on Software Reliability Engineering* (ISSRE 2000), IEEE, New York (2000).
5. J. Offutt and S. Liu, "Generating Test Data from SOFL Specifications," *The Journal of Systems and Software* **49**, No. 1, 49–62 (1999). See http://isse.gmu.edu/faculty/ofut/rsrch/spec.html.
6. L. Apfelbaum and J. Doyle, "Model-Based Testing," *Proceedings of the 10th International Software Quality Week* (*QW97*), Software Research, Inc., San Francisco, CA (1997).
7. A. Hartman and K. Nagin, "TCBeans, Software Test Toolkit," *Proceedings of the 12th International Software Quality Week* (*QW99*), Software Research, Inc., San Francisco, CA (1999).
8. J. M. Clarke, "Automated Test Generation from a Behavioural Model," *Proceedings of the 11th International Software Quality Week* (*QW98*), Software Research, Inc., San Francisco, CA (1998).
9. R. M. Poston, *Automated Testing from Object Models*, Aonix White Paper, Aonix, San Diego, CA (1998).

10. J. Grabowski, R. Scheurer, and D. Hogrefe, *Comparison of an Automatically Generated and a Manually Specified Abstract Test Suite for the B-ISDN Protocol SSCOP*, Technical Report A-97-07, University of Lubeck, Lubeck, Germany (1997).
11. U. Uyar, M. Fecko, A. Sethi, and P. Amer, "Generation of Realizable Conformance Tests Under Timing Constraints," *Proceedings of IEEE Military Communications Conference* (*MILCOM'98*), Boston, MA (October 1998).
12. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol Verification as a Hardware Design Aid," *IEEE International Conference on Computer Design: VLSI in Computers and Processors* (1992).
13. H. S. Hong, Y. G. Kim, S. D. Cha, D. H. Bae, and H. Ural, "A Test Sequence Selection Method for Statecharts," *Software Testing, Verification and Reliability* **10**, No. 4, 203–227 (2000).
14. G. Rothermel, M. J. Harrold, and A. Safe, "Efficient Regression Test Selection Techniques," *ACM Transaction on Software Engineering and Methodology* **6**, No. 2, 173–210 (1997).
15. G. Wimmel, H. Lutzbeyer, A. Pretschner, and O. Slotosch, "Specification Based Test Sequence Generation with Propositional Logic," *Software Testing, Verification and Reliability* **10**, No. 4, 229–248 (2000).
16. A. Aharon, A. Gluska, L. Fournier, Y. Lichtenstein, and Y. Malka, *Method for Measuring Architectural Test Coverage for Design Verification and Building Conformal Test*, U.S. Patent No. 5,724,504.
17. B. Marick, *The Craft of Software Testing*, Prentice Hall, Englewood Cliffs, NJ (1995).
18. *Portable Operating System Interface*, IEEE standard 1003.1, The Institute of Electrical and Electronics Engineers (1990).
19. IEEE POSIX Certification Authority, http://standards.ieee.org/regauth/posix/.
20. O. Agesen and D. Detlefs, "Finding References in Java Stacks," *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, P. Dickman and P. R. Wilson, Editors, ACM, New York (October 1997).
21. S. Sinha and M. J. Harrold, "Analysis and Testing of Programs with Exception-Handling Constructs," *IEEE Transactions on Software Engineering* **26**, No. 9, 849–871 (2000).
22. G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York (1979).
23. D. M. Woit, *Realistic Expectations of Random Testing*, Technical Report, McMaster University, Hamilton, ON, Canada (May 1992).
24. C. Kaner, "Software Negligence and Testing Coverage," *Proceedings of STAR 96: The Fifth International Conference on Software Testing Analysis and Review*, Orlando, FL (May 1996), pp. 299–327.
25. R. H. Carver and K-C Tai, "Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs," *IEEE Transactions on Software Engineering* **24**, No. 6, 471–490 (June 1998).
26. J. A. Whittaker and M. G. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Transactions on Software Engineering* **20**, No. 10, 249–262 (October 1994).
27. D. Gelperin, "Ultra-Understandable Decision Tables," available with the U3 Modeling Resource Kit at http://StickyMinds.com. Search the site using string U3. Posted October, 2000.
28. A. Bertolino and F. Basanieri, "A Practical Approach to UML-Based Derivation of Integration Tests," *4th International Software Quality Week Europe and International Internet Quality Week Europe* (*QWE 2000*), Brussels, Belgium (November 2000).
29. S. R. Dala, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-Based Testing in Practice," *21st International Conference on Software Engineering* (*ICSE'99*), Los Angeles, CA (1999).
30. C. Williams and A. Paradkar, "Efficient Regression Testing of Multi-Panel Systems," *Tenth International Symposium on Software Reliability Engineering* (*ISSRE'99*), Boca Raton, FL (November 1999).
31. A. Paradkar, "SALT—An Integrated Environment to Automate Generation of Function Tests for APIs," *11th International Symposium on Software Reliability Engineering* (*ISSRE'00*), San Jose, CA (2000), pp. 304–316.
32. O. Henniger and H. Ural, "Test Generation Based on Control and Data Dependencies within Multi-Process SDL Specifications," *Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC* (*SAM2000*), Grenoble, France (June 2000).
33. J. R. Horgan, S. London, and M. R. Lyu, "Achieving Software Quality with Testing Coverage Measures," *IEEE Computer* **27**, No. 9, 60–69 (1994).
34. M. Schmitt, M. Ebner, and J. Grabowski, "Test Generation with Autolink and Test-Composer," *Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC* (*SAM2000*), Grenoble, France (June 2000).
35. J. Hartmann, C. Imoberdorf, and M. Meisinger, "UML-Based Integration Testing," *Proceedings of the International Symposium on Software Testing and Analysis*, Portland, OR, ACM, New York (2000), pp. 60–70.
36. T. Quatrani and G. Booch, *Visual Modeling with Rational Rose 2000 and UML*, Addison-Wesley Publishing Co., New York (1999).
37. A. Gargantini and C. Heitmeyer, "Using Model-Checking to Generate Tests from Requirements Specifications," *Proceedings of the 7th European Software Engineering Conference* (*7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*), Toulouse, France, September 1999, Springer-Verlag, New York (1999).
38. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computations*, Addison-Wesley Publishing Co., New York (1979).
39. Strictly speaking, the system under test is compliant with the Open Group UNIX 95 standard, which is a super set of the POSIX standard.
40. Ann T. Totten from the IBM Poughkeepsie Laboratory reported these results at an IBM symposium on software testing, in 2000.
41. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Sunsoft Java Series, Addison-Wesley Publishing Co., New York (1997).
42. B. Jacobs, *A Formalisation of Java's Exception Mechanism*, Technical Report CSI-R0015, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands (1997). Also in the *Proceedings of the European Symposium on Programming*, Genova, Italy (April 2001).
43. S. Ntafos, "A Comparison of Some Structural Testing Strategies," *IEEE Transactions on Software Engineering* **14**, No. 6, 868–874 (1988).
44. P. Frankl and E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering* **14**, No. 10, 1483–1498 (1998).
45. J. W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," *IEEE Transactions on Software Engineering SE-9 (1983)*, pp. 347–354.
46. K. Barabash, N. Buchbinder, T. Domani, E. K. Kolodner, Y. Ossia, S. S. Pinter, J. Shepherd, R. Sivan, and V. Uman-

ski, "Mostly Accurate Stack Scanning," *First Java Virtual Machine Research and Technology Symposium*, Monterey, CA (April 2001).
47. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño Virtual Machine," *IBM Systems Journal* **39**, No. 1, 211–238 (2000).
48. J. M. Stichnoth, G.-Y. Lueh, and M. Cierniak, "Support for Garbage Collection at Every Instruction in a Java Compiler," *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, Atlanta, GA (May 1999), pp. 118–127.
49. O. Agesen, D. Detlefs, and J. E. Moss, "Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines," *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, Montreal, Canada (June 1998), pp. 269–279.
50. *IBM C and C++ Maintenance and Test Toolsuite*, User's Manual, Release 1.1, IBM Corporation (incorporates *χsuds* technology from Bellcore).
51. I. Somerville, *Software Engineering*, Addison-Wesley Publishing Co., New York (1998). Section 29.1.

Accepted for publication August 27, 2001.

Eitan Farchi *IBM Research Division, Haifa Research Laboratory, MATAM, Haifa 31905, Israel (electronic mail: farchi@il.ibm.com).* Dr. Farchi received his Ph.D. degree in mathematics from the University of Haifa, Israel, in 1999. He also holds a B.Sc. degree in mathematics and computer science and an M.Sc. degree in mathematics from the Hebrew University, Jerusalem, Israel. Since 1992 he has been with the IBM Haifa Research Laboratory, where he led an effort toward improving the performance of operating systems. He is currently involved in software testing and in developing coverage-directed tools for testing concurrent and distributed programs. Dr. Farchi is a frequent speaker at software testing conferences, is the author of a tutorial on the testing of distributed components, and teaches software engineering at the University of Haifa.

Alan Hartman *IBM Research Division, Haifa Research Laboratory, MATAM, Haifa 31905, Israel (electronic mail: hartman@il.ibm.com).* Dr. Hartman is a mathematician working in the Verification Technologies Department at the IBM Haifa Research Laboratory. He received his Ph.D. degree in 1980 at the University of Newcastle, Australia, in the field of combinatorial design theory. He has worked at the University of Waterloo, the University of Toronto, and at Telstra Research Laboratories. His research interests include combinatorics, graph theory, algorithms, software and hardware verification, and communication networks design.

Shlomit S. Pinter *IBM Research Division, Haifa Research Laboratory, MATAM, Haifa 31905, Israel (electronic mail: shlomit@il.ibm.com).* Dr. Pinter is a research staff member in the Systems and Software Department at the IBM Research Laboratory in Haifa, and an adjunct professor with the Computer Science Department at the University of Haifa. For her Ph.D. in computer science, awarded to her by Boston University in 1984, she wrote a thesis in the area of distributed computing. Before joining IBM in 1996, she was with the Department of Electrical Engineering at the Technion, Israel Institute of Technology (1983–1995). She was a research specialist with the Laboratory for Computer Science at MIT, Cambridge, MA (1978–1981), and a visiting scholar at Yale and Stanford Universities (1988–1989 and 1997–1998, respectively). Dr. Pinter has published extensively in international journals and conference proceedings. Her research interests include compilation techniques for advanced computer architectures, program analysis and parallelization, and distributed and parallel computing. Dr. Pinter is a member of the editorial board of the *International Journal of Parallel Programming*.