Generating Tests from UML Specifications

Jeff Offutt and Aynur Abdurazik *

George Mason University, Fairfax VA 22030, USA

Abstract. Although most industry testing of complex software is conducted at the system level, most formal research has focused on the unit level. As a result, most system level testing techniques are only described informally. This paper presents a novel technique that adapts pre-defined state-based specification test data generation criteria to generate test cases from UML statecharts. UML statecharts provide a solid basis for test generation in a form that can be easily manipulated. This technique includes coverage criteria that enable highly effective tests to be developed. To demonstrate this technique, a tool has been developed that uses UML statecharts produced by Rational Software Corporation's Rational Rose tool to generate test data. Experimental results from using this tool are presented.

1 Introduction

There is an increasing need for effective testing of software for complex safety-critical applications, such as avionics, medical, and other control systems. These software systems usually have clear high level descriptions, sometimes in formal representations. Unfortunately, most system level testing techniques are only described informally. This paper is part of a project that is attempting to provide a solid foundation for generating tests from system level software specifications via new coverage criteria. Formal coverage criteria offer testers ways to decide what test inputs to use during testing, making it more likely that the testers will find any faults in the software and providing greater assurance that the software is of high quality and reliability. Such criteria also provide stopping rules and repeatability.

Although UML provides a powerful mechanism for describing software that is safety-critical or that must be highly reliable, it still is not in widespread use. One advantage of using software description languages such as UML is that they provide a convenient basis for selecting tests. The purpose of this research project is to take advantage of UML to produce highly effective software system-level tests. Although UML is not completely formalized, certain aspects of the language are precise enough to be utilized for test generation, in particular the UML Statecharts.

The research presented in this paper is part of a long term project that is looking at ways to generate tests from specifications. This paper presents formal

* This work is supported in part by the U.S. National Science Foundation under grant CCR-98-04111 and in part by Rockwell Collins, Inc.

criteria for developing test inputs from UML statecharts. An overview of software testing is given in the next section, then the criteria are described, algorithms and a proof-of-concept tool are presented, and finally results from an empirical evaluation are given.

2 Software Testing

Software testing includes executing a program on a set of test cases and comparing the actual results with the expected results. Testing and test design, as parts of quality assurance, should also focus on fault prevention. To the extent that testing and test design do not prevent faults, they should be able to discover symptoms caused by faults. Finally, tests should provide clear diagnoses so that faults can be easily corrected [3].

This paper uses the following definitions. A *test* or *test case* is a general software artifact that includes test case input values, expected outputs for the test case, and any inputs that are necessary to put the software system into the state that is appropriate for the test input values. A test specification language (TSL) is a language that can be used to describe all components of a test case. The components considered here are *test case values*, *prefix values*, *verify values*, *exit commands*, and *expected outputs*. Test case values directly satisfy the test requirements, and the other components supply supporting values. A *test case value* is the essential part of a test case, which comes from the test requirements. It may be a command, user inputs, or software function and values for its parameters. In state-based software, test case values are usually derived directly from triggering events and preconditions for transitions. A test case *prefix value* includes all inputs necessary to place the software system into the appropriate state for running the test case values. Any inputs that are necessary to show the results are *verify values*, and *exit commands* terminate the execution of the software. *Expected outputs* are the outputs of the test case on a correct version of the software.

Test requirements are specific things that must be satisfied or covered during testing, for example, reaching statements are the requirements for statement coverage. *Test specifications* are specific descriptions of test cases, often associated with test requirements or criteria. For statement coverage, test specifications are the conditions necessary to reach a statement. A *testing criterion* is a rule or collection of rules that impose test requirements on a set of test cases. A *testing technique* guides the tester through the testing process by including a testing criterion and a process for creating test case values.

Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*, which is the percent of requirements that are satisfied. There are various ways to classify adequacy criteria. One of the most common is by the source of information used to specify testing requirements and in the measurement of test adequacy. Hence, an adequacy criterion can be specification-based or program-based.

A *specification-based* criterion specifies the required testing in terms of identified features of the specifications of the software, so that a test set is adequate if all the identified features have been fully exercised. Here the specifications are used to produce test cases, as well as to produce the program. A *program-based* criterion specifies testing requirements in terms of the program under test and decides if a test set is adequate according to whether the program has been thoroughly exercised. For example, if the criterion of branch testing is used, the tests are required to cover each branch in the program.

There are two main roles a specification can play in software testing [9]. The first is to provide the necessary information to check whether the output of the program is correct [8]. Checking the correctness of program outputs is known as the *oracle problem*. The second is to provide information to select test cases and to measure test adequacy [12].

Specification-based testing (SBT) offers many advantages in software testing. The (formal) specification of a software product can be used as a guide for designing functional tests for the product. The specification precisely defines fundamental aspects of the software, while more detailed and structural information is omitted. Thus, the tester has the essential information about the product's functionality without having to extract it from inessential details.

Formal specifications provide a simpler, structured, and more formal approach to the development of functional tests than using non-formal specifications. One significance of producing tests from specifications is that the tests can be created earlier in the development process, and be ready for execution **before** the program is finished. Additionally, when the tests are generated, the test engineer will often find inconsistencies and ambiguities in the specifications, allowing the specifications to be improved before the program is written.

3 Test Data Generation Techniques Based on UML

Offutt has previously developed criteria for generating test data from SCR specifications [7] and from SOFL specifications [6]. These techniques have been adapted to UML Statecharts, and a tool has been built that automatically generates tests from UML statecharts created using Rational Software Corporation's Rational Rose tool [4].

UML can be used to specify a wide range of aspects of a system. UML statecharts are based on finite state machines using an extended Harel state chart notation, and are used to represent the behavior of an object. As they are the most formalizable aspects of UML, statecharts provide a natural basis for test data generation.

The *state* of an object is the combination of all attribute values and objects the object contains. The dynamics of objects are modeled through transitions among states. The UML syntax for transitions is:

```
event-name (parameters) [guard] / action list ^ event list
```

`Event-name` is a label for the transition, and `parameters` are the variables associated with the event that must be triggered for the transition to be taken place. `Guard` is a precondition that controls whether the transition is taken or not, and `action list` and `event list` define changes in the software that occur as a result of the transition.

UML categorizes transitions into five types: *high-level transitions*, *compound transitions*, *internal transitions*, *completion transitions*, and *enabled transitions*. This research is only interested in enabled transitions. The previous testing model was based primarily on predicate satisfaction. In UML, the enabled transitions are similar to transitions that are based on the notion of *predicate satisfaction*. An *enabled transition* is enabled by an event, and it originates from an active state. An enabled transition is triggered when there exists at least one full path from the source state to the target state.

Four kinds of events can be specified in UML: call events, signal events, time events, and change events. A *call event* represents the reception of a request to synchronously invoke a specific operation. A *signal event* represents the reception of a particular (synchronous) signal. A *time event* represents the passage of a designated period of time after a designated event (often the entry of the current state) or the occurrence of a given date and time. A *change event* models an event that occurs when an explicit boolean expression becomes true as a result of a change in value of one or more attributes or associations.

Since change events can be expressed as predicates, these are used as the basis for generating tests. A change event is raised implicitly when the associated predicate becomes true. Change events are different from guards. A guard is only evaluated when an event is dispatched whereas, conceptually, the boolean expression associated with a change event is evaluated continuously until it becomes true. The event that is generated remains until it is consumed even if the boolean expression changes to false. In UML, a change event is modeled by using the keyword *when* followed by a Boolean expression. Figure 1 illustrates a change event. This change event indicates that the function `selfTest()` is called at 11:49 PM.

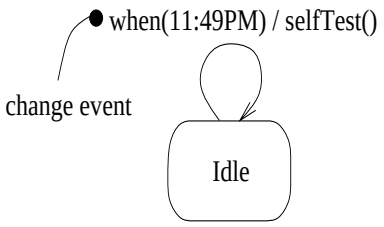

Fig. 1. Change Events

Change event enabled transitions are used to define four levels of testing: (1) the transition coverage level, (2) the full predicate coverage level, (3) the

transition-pair coverage level, and (4) the complete sequence level.

It is possible to apply all levels, or to choose a level based on a cost/benefit tradeoff. The first two are related; the transition coverage level requires many fewer test cases than the full predicate coverage level, but if the full predicate coverage level is used, the tests will also satisfy the transition coverage level (full predicate coverage *subsumes* transition coverage). Thus only one of these two should be used. The latter two levels are meant to be independent; transition-pair coverage is intended to check the interfaces among states, and complete sequence testing is intended to check the software by executing the software through complete execution paths. As it happens, transition-pair coverage subsumes transition coverage, but they are designed to test the software in very different ways.

3.1 Transition Coverage Level

It seems reasonable to expect that to test the software adequately, the tester should at minimum use tests that cause every transition in every statechart to be taken. This level requires just that, by requiring test cases that satisfy each precondition in the specification at least once. In the criteria definitions, T is a set of test cases, and SG is a *specification graph*, a graph that represents the transitions in a statechart. Although the tests are intended to be executed on an implementation of the specification, we say that a test *traverses* a transition to indicate that, from a modeling perspective, the test causes the transition's predicate to be true, and the implementation will change from the transition's pre-state to its post-state.

Coverage Level 1, Transition Coverage: *the test set T must satisfy every transition in the SG.*

3.2 Full Predicate Coverage Level

Small inaccuracies in the specification predicates can lead to major problems in the software. The full predicate coverage level takes the philosophy that to test the software, we should at least provide inputs to test each clause in each predicate. This level requires that each clause in each predicate on each transition is tested independently, thus attempting to address the question of whether each clause is necessary and is formulated correctly. This paper follows the definitions in DO178B [10]. The Boolean operators are AND, OR, and NOT. A *clause* is a Boolean expression that contains no Boolean operators. For example, relational expressions and Boolean variables are clauses. A *predicate* is a Boolean expression that is composed of clauses and zero or more Boolean operators. A predicate without a Boolean operator is also a clause. If a clause appears more than once in a predicate, each occurrence is a distinct clause. (DO178B uses the terms "condition" and "decision", but the more common "clause" and "predicate" are used here.)

Full predicate coverage is based on the philosophy that each clause should be tested independently, that is, while not being influenced by the other clauses. In other words, each clause in each predicate on every transition must independently affect the value of the predicate.

Coverage Level 2, Full Predicate Coverage: *for each predicate P on each transition, T must include tests that cause each clause c in P to result in a pair of outcomes where the value of P is directly correlated with the value of c.*

In this definition, "directly correlated" means that c controls the value of P, that is, one of two situations occurs. Either c and P have the same value (c is true implies P is true and c is false implies P is false), or c and P have opposite values (c is true implies P is false and c is false implies P is true). This explicitly disallows cases such as c is true implies P is true and c is false implies P is true.

Note that if full predicate coverage is achieved, transition coverage will also be achieved. To satisfy the requirement that the *test clause* controls the value of the predicate, other clauses in the predicate must be either `True` or `False`. For example, if the predicate is $(X \wedge Y)$, and the test clause is X, then Y must be `True`. Likewise, if the predicate is $(X \vee Y)$, Y must be `False`. More details, including how this can be accomplished with general predicates, are given elsewhere [6, 7].

3.3 Transition-Pair Coverage Level

The previous testing levels test transitions independently, but do not test sequences of state transitions. This level requires that pairs of transitions be taken.

Coverage Level 3, Transition-pair Coverage: *For each pair of adjacent transitions $S_i : S_j$ and $S_j : S_k$ in SG, T contains a test that traverses the pair of transitions in sequence.*

3.4 Complete Sequence Level

It seems very unlikely that any successful test method could be based on purely mechanical methods; at some point the experience and knowledge of the test engineer must be used. Particularly at the system level, effective testing probably requires detailed domain knowledge. A *complete sequence* is a sequence of state transitions that form a complete practical use of the system. In most realistic applications, the number of possible sequences is too large to choose all complete sequences. In many cases, the number of complete sequences is infinite.

Coverage Level 4, Complete Sequence: *The test engineer must define meaningful sequences of transitions on the statechart diagram by choosing sequences of states that should be entered.*

4 A Rose-based Test Data Generation Tool

It is possible to automate almost all of the steps of generating test data for these criteria. If a machine-readable form of the specifications is available, the transition conditions can be read directly. Test requirements take the form of partial truth tables defined on state transition predicates and pairs of state transition predicates. Given a formal functional specification, most if not all of these test requirements can be generated automatically. The most complicated part of the test case is the test prefix, which includes inputs necessary to put the system into a particular pre-state. Test prefixes are handled by creating a specification graph, which contains all the states and their transition relationships. Prefixes can then be generated automatically by walking through the graph.

To evaluate these criteria, we have developed a proof-of-concept test data generation tool. This tool, UMLTest, is integrated with the Rational Rose case tool [4]. Rose saves UML specifications in a plain text format, which can be easily read. UMLTest generates test cases at full predicate and transition-pair levels. Transition coverage is subsumed by full predicate coverage, and complete sequence coverage is not fully automatable.

UMLTest parses a Rose specification file (called an MDL file) to get the semantic meanings of the specifications. MDL files store specification information from different perspectives. There are two main categories of information, logical and physical. The specification itself is grouped into two packages: *use case*s and *object collaboration diagrams* are packaged into *Use Case Package*s, and *class diagrams* and *state transition diagrams* are packaged into *Logical Views*. Figure 2 shows the internal structure of the class diagram and state transition diagram in an MDL file.

UMLTest makes several assumptions about the UML specification file input. It relies on the Object Constraint Language (OCL), which is an expression language that enables constraints to be described on object-oriented models and other modeling artifacts [11]. A *constraint* is a restriction on one or more values within an object-oriented model or system. OCL is part of UML and is the standard for specifying invariants, preconditions, postconditions, and other kinds of constraints. UMLTest makes several assumptions:

- All transitions are triggered by change events.
- Events and conditions are expressed through boolean type class attributes.
- The specification is written strictly following the UML notation. For example, *when* denotes a change event and conditions are in solid brackets ([]). Because there is no way to check whether a specification is well-formed or consistent, this assumption cannot be checked. The OCL does not have a mechanism to enforce its syntactic rules on all parts of the UML specification. Also, Rose does not have a function to write the specification in OCL.
- State transitions are deterministic.

Figure 3 is a UML class diagram describing UMLTest. Classes are represented as boxes that have three parts, the class name, data members that are declared in the class, and methods of the class. The main entry point (UMLTest)

```
Logical Models
    object Class
        classAttributes

        -------------- State Transition: Logical -------------

        State Machine
            object State /* StartState, Normal, EndState */
                State Transition
                State Machine
                    Object State /* Normal */

        -------------- State Transition: Physical ------------

        State Diagram
            State View /* StartState, Normal, EndState */
            Transition View

    object Association
        object Role
Logical Presentations
    object ClassDiagram /* with grouping and name */
        object ClassView
            Association View
            Role View
            Inheritance View
```

Fig. 2. Structure of MDL File for Class Diagrams and State Transition Diagrams

has three objects, (1) a UML specification parser, (2) a full predicate test case generator, and (3) a transition-pair test case generator.

`UMLSpecParser` reads a UML specification text file, parses it, and generates state transition table(s) for classes that have state machines. `FullPredicate` takes a state transition table as an input, generates test cases for the full predicate coverage criterion, and saves the test cases into an ASCII text file. `TransitionPair` takes a state transition table as input, generates test cases for the transition-pair coverage criterion, and saves the test cases in an ASCII text file.

4.1 Algorithms

This section presents two algorithms used in UMLTest. Algorithms were developed to generate test cases for the *full predicate coverage*, *transition coverage*,

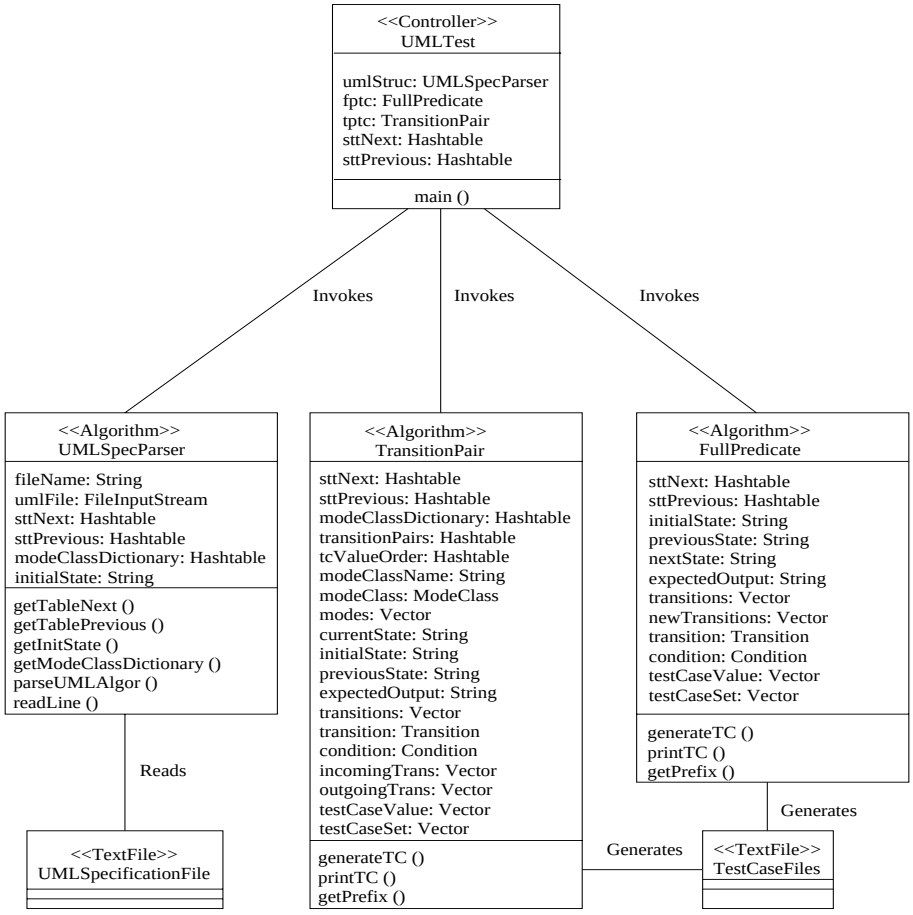

Fig. 3. Class Diagram for UMLTEST **Tool**

and *transition-pair coverage* criteria. A *prefix generation* algorithm was used in test data generation algorithms to create the values necessary to reach a particular state. Because of space limitations, only the full predicate coverage and prefix generation algorithms are presented. The additional algorithms are available in a technical report [1].

Get Prefix Algorithm. Figure 4 gives an algorithm for generating test prefix values from a specification graph. The input is a state (the *test state*) in the graph, and it finds a path from an initial state in the graph to the test state.

Generate Full-Predicate Coverage Test Cases Algorithm. Figure 5 gives an algorithm for generating test cases for the *full predicate coverage* criterion. Algorithm *GenerateFullPredicateCoverageTCs* takes a *state transition table* as

```
algorithm:        GetPrefix (State)
input:            Test state of a transition.
output:           Inputs to get to the given state.
output criteria:  No redundant inputs.
declare:          prefix (s) -- Inputs to reach state s.
                  incomingTrans (s) -- The set of incoming transitions.
                  event (otr) -- Trigger event for transition otr.
                  whenCondition (otr) -- Precondition for otr.
                  nextState (otr) -- Next state for transition otr.
                  expectedOutput -- Post-state after transition.
                  TCValue (otr) -- Value assignments for the trigger
                  event and when condition variables for otr.

GetPrefix (State)
BEGIN -- Algorithm GetPrefix
    s = State
    prefixStates = prefixStates ∪ s
    WHILE (s IS NOT initial state) LOOP
        get incomingTrans (s)
        prefix (s) = EMPTY
        IF (∃ transition itr ∈ incomingTrans (s) such that
        prevState (itr) = initialState) THEN
            s = prevState (itr)
            prefixStates = prefixStates ∪ s
            EXIT
        ELSE
            s = prevState (itr) such that itr ∈ incomingTrans (s) ∧
                prevState (itr) ∉ prefixStates
            prefixStates = prefixStates ∪ s
        END IF
    END LOOP
END Algorithm GetPrefix
```

Fig. 4. The GetPrefix Algorithm

input, and generates test cases for the full predicate coverage criterion. It processes each outgoing transition of each source state, generates a test case that makes the transition valid, and then generates test cases that make the transition invalid. When generating a test case, `GetPrefix ()` is used to obtain prefixes to reach the source state of a transition. Then each variable in the transition predicate is assigned a test case value. To avoid redundant test case value assignments, those variables that already have assigned values in the prefixes are not considered in the test case value assignment process. After all test case values are generated, an additional algorithm is run on the test cases to identify and remove redundant test cases.

```
algorithm:          GenerateFullPredicateCoverageTCs (STTable)
input:              State transition table.
output:             Test cases for full predicate coverage.
output criteria:    Test cases contain prefix, test case values,
                    and expected output.
assumption:         Clauses are disjunctive.
                    No redundant assignments in prefix and test cases.
declare:            prefix (s) -- Inputs to get to the state s.
                    outgoingTrans (s) -- Set of outgoing transitions.
                    event (otr) -- Trigger event for transition otr.
                    whenCondition (otr) -- Precondition for otr.
                    nextState (otr) -- Next state for transition otr.
                    expectedOutput -- Post-state after transition.
                    TCValue (otr) -- Value assignments for the trigger
                    event and when condition variables for otr.

GenerateFullPredicateCoverageTCs (STTable)
BEGIN -- Algorithm GenerateFullPredicateCoverageTCs
   TestCaseSet = EMPTY
   FOR EACH source state s in STTable
      prefix (s) = GetPrefix (s)
      get outgoingTrans (s)
      -- Generate one test case for each transition
      FOR EACH outgoing transition otr ∈ outgoingTrans (s)
         expectedOutput = nextState (otr)
         TCValue (otr) = EMPTY
         get event (otr) and whenConditions (otr)
         -- Check for redundancy
         IF (¬ ∃ a condition variable var ∈ prefix (s) s.t.
         var.name = event (otr).name ∧
         var.value = event (otr).value)
            TCValue (otr) = TCValue (otr) ∪
            {(event (otr).name, event (otr).beforeValue)}
         END IF
         -- Assign value for clauses in when condition
         FOR EACH clause_i in whenConditions (otr)
            IF (¬ ∃ a condition variable var ∈ prefix (s) s.t.
            var.name = clause_i.name ∧ var.value = clause_i.value)
               TCValue (otr) = TCValue (otr) ∪
                 {(clause_i.name, clause_i.value)}
            END IF
         END FOR
         TCValue (otr) = TCValue (otr) ∪ {(event (otr).name,
            event (otr).afterValue)}
         TestCaseSet = TestCaseSet ∪ {(prefix (s), TCValue (otr),
            expectedOutput}
```

Fig. 5. The GenerateFullPredicateCoverageTCs Algorithm

```
            -- get test cases for invalid transitions
            expectedOutput = current state s
            FOR EACH variable var in TCValue (otr)
               TCValue (otr) = TCValue (otr) - {(var.name, var.value)}
               var.value = ¬var.value
               TCValue (otr) = TCValue (otr) ∪ {(var.name, var.value)}
               TestCaseSet = TestCaseSet ∪ {(prefix (s),
                   TCValue (otr), expectedOutput)}
            END FOR
         END FOR
      END FOR
   END Algorithm GenerateFullPredicateCoverageTCs
```

Fig. 5. The GenerateFullPredicateCoverageTCs Algorithm - continued

5 Empirical Evaluation

An empirical study has been undertaken to demonstrate the feasibility of these criteria. The goal was to demonstrate that the specification-based criteria can be effectively used. Tests were created and then measured on the basis of their fault-detection abilities. One moderate size program was used (cruise control [2, 5]), representative faults were seeded, and test cases were generated by UMLTest.

UMLTest generated 54 full predicate test cases; after redundant test cases were eliminated, 34 remained to be used during testing. UMLTest generated 34 test cases for transition-pair coverage.

A model of the cruise control problem was implemented in about 400 lines of C. `Cruise` has seven functions, 184 blocks, and 174 decisions. Twenty-five faults were created by hand and were inserted into the program by creating separate versions. Most of these faults are small modifications (such as changing a variable name or arithmetic operator), and most were in the logic that implemented the state machine. Four were naturally occurring faults, made during initial implementation.

For comparison, tests were created to satisfy statement coverage. These tests were generated by hand to execute every statement in the (original) implementation at least once, and 27 tests were needed. To eliminate bias, these tests were not generated by either author.

Figure 6 shows the fault coverage percentage for full predicate, transition-pair, and statement coverage criteria test cases. The full predicate tests were able to find all the faults, and the transition-pair tests did significantly better than the statement coverage tests.

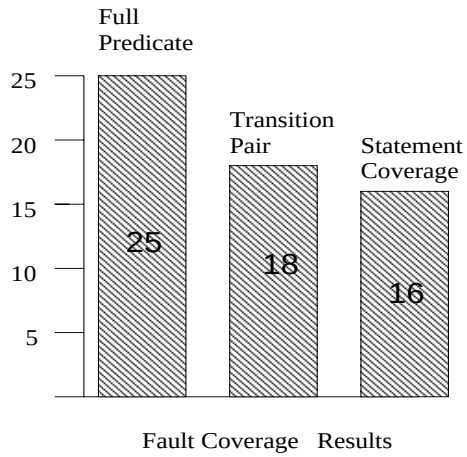

Fig. 6. Test Data Results

6 Conclusions

This paper presented three results. First, a novel collection of testing criteria for UML was described. These criteria allow software system-level tests to be derived from UML statechart diagrams. As far as we know, this is the first formal testing technique that is based on UML. Second, a proof-of-concept automatic test data generation tool has been developed. This solves one of the most important problems in software testing – developing the actual tests. This tool, UMLTest, is integrated with the Rational Rose tool and is completely automated. This is the first tool that we know of that can automatically generate tests from UML specifications. Third, empirical results from using UMLTest to evaluate the testing criteria were given. The results indicate that highly effective tests can be automatically generated for system level testing. This provides strong motivation for using UML, and raises the possibility of developing software that is more reliable than what is being developed by current means.

There are a number of limitations of this current work. The most important is that tests were derived from statecharts and enabled transitions; other aspects of UML have not yet been addressed. By not using other types of transitions, certain states may never be entered. We are currently extending our test criteria to incorporate other types of transitions. We also hope to incorporate other parts of UML if the semantics can be sufficiently formalized, including UML class models. Although the test model is general, the current tool requires all variables to be boolean. Extending the tool to generate values for variables of other types is straightforward, but tedious, so has not been done for this research tool. Finally, the empirical study was limited to one program and one set of tests for each testing technique. We are currently working on extending this study to compare with multiple programs.

References

1. Aynur Abdurazik and Jeff Offutt. Generating test cases from UML specifications. Technical report ISE-TR-99-105, Department of Information and Software Engineering, George Mason University, Fairfax VA, 1999. http://www.ise.gmu.edu/techrep.
2. J. M. Atlee. Native model-checking of SCR requirements. In *Fourth International SCR Workshop*, November 1994.
3. B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.
4. Rational Software Corporation. *Rational Rose 98: Using Rational Rose*. Rational Rose Corporation, Cupertina CA, 1998.
5. Zhenyi Jin. Deriving mode invariants from SCR specifications. In *Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems*, pages 514–521, Montreal, Canada, October 1996. IEEE Computer Society.
6. Jeff Offutt and Shaoying Liu. Generating test data from SOFL specifications. *The Journal of Systems and Software*, 1999. To appear.
7. Jeff Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, Las Vegas, NV, October 1999. IEEE Computer Society Press.
8. Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.
9. D. J. Richardson, O. O'Malley, and C. Tittle. Approaches to specification-based testing. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 86–96, Key West, Florida, December 1989. ACM SIGSOFT'89.
10. RTCA Committee SC-167. Software considerations in airborne systems and equipment certification, Seventh draft to Do-178A/ED-12A, July 1992.
11. Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley, 1999. ISBN 0-201-37940-6.
12. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

This article was processed using the LaTeX macro package with LLNCS style