
Ryan Voigt, Kareem Fazal, Hassan Reza
School of Aerospace Sciences
University of North Dakota, Grand Forks, USA
reza@cs.und.edu

Abstract

As the most formalizable UML diagram, statecharts make a natural basis for specification based test generation. In this paper we discuss a specification-based testing method that extends specification-based testing method known as Testing Flow Graphs (TFG). Our extended TFG method allows test sequences to be generated that also meet full predicate coverage. To show the effectiveness and efficiency of our approach relative to the original work, we carried out an empirical evaluation using mutation analysis.

1. Introduction

Specification-based testing confers a number of advantages to the software development process. A specification provides an exact description of the software’s fundamental aspects while excluding more detailed information. This allows a tester to extract the product’s basic functionality without wading through inessential details. By deriving tests from the software specification, tests can be produced before the software itself. Since many faults occur during the design phase, early identification of them can reduce total development times and costs. In addition, developing tests forces a detailed look at the specification itself, which may reveal ambiguities and/or inconsistencies. These can then be fixed early in the development cycle at a minimum of cost.

The Unified Modeling Language (UML) [1] is a visual modeling language that can be used to describe a wide range of aspects of software systems. UML has become the de facto industry standard for modeling large, complex software systems. These models provide a convenient basis for selecting tests, but are often evaluated manually using inspections and walkthroughs. Such informal methods are tedious, requiring the reviewer to track large amounts of information, and lack the rigor found in more systematic testing techniques. Instead, executable models can be used to conduct dynamic design testing to enhance fault detection. Unfortunately, UML is not fully formalized. Of its nine types of diagrams statecharts are the most formalizable, making them a natural basis for test generation.

Statecharts are finite-state machines extended with hierarchy and orthogonality. They are comprised of states, transitions, events and actions that emphasize the flow of control from one state to another. Finite-state machines best describe the dynamic behavior of a system, and finite-state model based testing has been studied extensively. UML statecharts can be used to represent the behavior of communication protocols, graphical user interfaces and other event-driven systems. Model-based testing approaches can automatically derive executable tests from UML statecharts, thus providing benefits like systematic testing and test adequacy. Since a key requirement of software testing is to ensure test adequacy, these features make statechart based testing very useful.

2. Related Works

Offutt and Aburazik [2] present a technique to generate test cases from UML statecharts without transformation. However their approach focused solely on change events and enabled transitions, which would potentially leave some sections of a statechart unexercised. Their work includes algorithms for full predicate and transition pair coverage, as well as a small but diverse empirical evaluation using mutation testing.

Murthy et al. [5] attempt to lay a new foundation for UML statechart testing by introducing their Test ready UML statechart, which are UML statecharts annotated with events, guard conditions, tasks and test statements along state transitions. Their models have an equivalent extended context-free grammar. Cycles in the graph are automatically
detected, marked and expanded during test case generation a random number of times up to a user specified limit. The algorithm to generate test cases for path coverage is included. No empirical evaluation was conducted.

The work by Kansomkeat and Riverpiboon [3] proposes an automatic testing technique in which they transform statechart diagrams into intermediate diagrams called the Testing Flow Graphs (TFG), which reduce the complexity. Their model consists of the g-nodes, g-transitions, s-nodes and the s-transitions, which represent the intermediate node, guard conditions, states and the events respectively. They explore the TFGs by analyzing the branches from the root node to each leaf node for state and transition coverage. They generate test sequences from the TFGs, which are dictated by the guard conditions and the sequence of events. S-node generates events which cause and do not cause a transition. Their events generate all the events which affirm as well as negate the guard conditions. They focused on Mutation Testing for the evaluation of their test cases, by injecting faults called Mutants into the original program. Their fault detection ability measures the effectiveness of the test case.

3. Method

Our initial work involved formalizing Kansomkeat and Riverpiboon's techniques for creating TFGs and generating the test sequences from them. While the methods were sound, they were presented in an intuitive, natural language approach, which left certain details unspecified or ambiguous. The authors had to infer details from the diagrams and fill in blanks through trial and error while working the examples themselves. Pseudocode for the original TFG generating algorithm is included in Appendix A. Several assumptions are included, but the authors feel it is very close to what Kansomkeat and Riverpiboon intended.

The TFGs were extended for full predicate coverage by changing g-transitions to represent a single predicate clause rather than an entire predicate. Sequences of g-nodes linked by g-transitions would then represent a given predicate, allowing the test sequence generating algorithm to access clauses for individual negation. Simple single clause predicates become the same TFG structure that they were in the original technique. The differences occur for compound predicates and shared clauses. Compound predicates are those that contain more than one clause. Figure 1 shows a fragment of a statechart that uses an event with a compound predicate and the TFG structures it becomes under the original and extended techniques. Compound predicates become a number of g-transitions equal to their number of clauses with g-nodes between g-transitions. Shared clauses are those that appear in multiple predicates. Figure 2 shows the statechart, original and revised TFG structures associated with shared clauses. Here again each clause has its own g-transition, but the common shared clauses need not be duplicated if the original predicates are constructed properly. In some cases, having multiple g-transitions for a given clause may be unavoidable, but this is not a problem since TFGs do not represent a flowchart like decision logic. Minimizing the number of g-transitions for a group of predicates merely reduces the amount of checking done by the test generation algorithm.
now allow the test sequence generation algorithm to achieve full predicate coverage with a few minor alterations. First, when a g-node is initially reached, all the g-transition paths off of it can be explored to collect every individual clause. These can then all be negated for the s-node’s negative testing. This result is identical to the negative test case produced by the original TFG test sequence algorithm, but by representing clauses individually, the algorithm can check to see if any clause is required to be both true and false, rendering the test case impossible. Second, when an individual predicate path is being tested, the individually represented clauses allows them to be cycled through all possible true/false combinations. This ensures that each clause independently affects the value of the predicate as it should. Finally the extended test sequence generation algorithm includes checks to remove logically impossible test inputs and test inputs generated to satisfy negative testing that are actually valid events. Pseudocode for the extended TFG generating algorithm is included in Appendix B.

4. Case Study

The case study used for the author’s method was a slightly contrived example of using a telephone whose local service has a complicated connection scheme. The statechart is shown in Figure 3. Events are shown in lower case with predicates in square brackets and individual clauses in capitals. In the Connect Local state, two of three checks must be passed before the call can be connected.

Figure 3. Statechart for dialing a telephone.

Figure 4 is the extended TFG created from the previous statechart. For comparison, the TFG produced by Kansomkeat and Rivepiboon’s original method is shown in Figure 5. The test sequences generated from the extended TFG are listed in Appendix C.
5. Empirical Evaluation

The authors evaluated their method using mutation analysis. Mutation analysis is a fault-based testing strategy that makes numerous small syntactic changes in a test program. These derivative programs are called mutants and are exercised with the same test sequences. Mutants are ‘killed’ if the test set is capable of causing behavioral differences between the mutant and the original program. The adequacy criteria for mutation analysis is the mutation score, which is the percentage of mutants killed by the test set.

A basic program shell of the telephone case study was coded in about 280 lines of Visual Basic, including extra controls used solely for testing. Twenty four mutants were created. Table 1 shows the mutant types and distribution. Of these mutants, Kansomkeat and Rivepiboon’s original method killed 21. The extended TFG approach killed 22. Both surviving mutants from the extended tests were of the same type, (deleted clause) and affected the same state. Closer examination revealed that the code had been written such that, for these particular mutants, control ‘fell through’ the logic exactly as it did in the test program, producing identical results.

<table>
<thead>
<tr>
<th>Mutant Type</th>
<th>Number of Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replace event name</td>
<td>4</td>
</tr>
<tr>
<td>Delete guard condition</td>
<td>4</td>
</tr>
<tr>
<td>Replace data name</td>
<td>4</td>
</tr>
<tr>
<td>Replace data value</td>
<td>4</td>
</tr>
<tr>
<td>Replace relational operator</td>
<td>4</td>
</tr>
<tr>
<td>Delete clause</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total Mutants</strong></td>
<td><strong>24</strong></td>
</tr>
</tbody>
</table>

Table 1. Mutant types and numbers.

6. Discussion

Since legs of a TFG stop when they come to an s-node that has already been expanded, TFGs cannot be used for path testing. This shortcoming could be overcome by tracking cycles in the graph and expanding them a certain number of times, but this would greatly balloon the size of the graph and compromise the TFG’s primary advantages: simplicity and conciseness. TFGs also currently lack support for concurrent states and state history mechanisms. These features could be the focus of future work.

The empirical evaluation was conducted using a single small program with relatively few mutants. More work would need to be done for increased confidence in the effectiveness of the TFG method, particularly with respect to other techniques.

7. Conclusions and future work

As in Kansomkeat and Rivepiboon’s original work, TFG based test sequence generation seems to be effective at identifying the faults in a small sample program. While more investigation is needed, TFGs currently appear efficient and, given the ease with which they were extended for full predicate coverage, adaptable. At the cost of slightly long test sequences, the extended technique adds the ability to detect a whole new family of faults.

8. References


APPENDIX A: Original TFG Generation Algorithm

```
buildTFG(currentState)
{
    if(currentState !in visitedStates)
    {
        add currentState to visitedStates
        FOR(each distinct event off current state)
        {
            create s-transition off current s-node using associated event
            IF(event has guard condition)
            {
                create g-node at end of s-transition
                FOR(each guard transition off current g-node)
                {
                    create g-transition off g-node using associated guard condition for event
                    create s-node at end of current transition for associated next state
                    CALL buildTFG(nextState)
                }
            }
            ELSE
            {
                create s-node at end of current transition for associated next state
                CALL buildTFG(nextState)
            }
        }
    }
}
```

APPENDIX B: Extended TFG Generation Algorithm

```
buildTFG(currentState or currentGNode)
{
    IF(currentState)
    {
        IF(currentState !in visitedStates)
        {
            add currentState to visitedStates
            FOR(each distinct event off current state)
            {
                create s-transition off current s-node using associated event
                IF(event has guard conditions)
                {
                    create g-node at end of s-transition
                    FOR(each distinct starting clause)
                    {
                        create g-transition off g-node using associated clause
                    }
                    create g-transition off g-node using associated clause
                }
            }
            ELSE
            {
                create s-node at end of current transition for associated next state
                CALL buildTFG(nextState)
            }
        }
    }
```
CALL buildTFG(currentGNode)
{
}
ELSE
{
create s-node at end of s-transition transition for associated next state
CALL buildTFG(nextState)
}
}
ELSE
{
IF(done with predicate branch)
{
create s-node at end of g-transition for associated next state
CALL buildTFG(nextState)
}
ELSE
{
create g-node at end of g-transition
FOR(each distinct clause from currentGNode)
{
create g-transition off current g-node using associated clause
CALL buildTFG(currentGNode)
}
}
}  

APPENDIX C: Extended TFG Test Sequences

Test Sequence #1
not lift
lift
not replace and not dial
dial

dial;;[valid][complete][local] //valid event
dial;;[invalid][complete][local] //impossible
dial;;[invalid][complete][local]
dial;;[valid][complete][local]
dial;;[invalid][complete][local]
dial;;[valid][complete][local]
dial;;[valid][complete][local]
dial;;[valid][complete][local]
dial;;[valid][complete][local]
dial;;[valid][complete][local]
dial;;[valid][complete][local]
dial;;[valid][complete][local]

Test Sequence #2
lift
dial
dial;;[valid][complete] //valid event
dial;;[valid][complete] //impossible ([local])
dial;;[valid][complete] //valid event
dial;;[valid][complete]

Test Sequence #3
lift
dial

Test Sequence #4
lift
dial
dial;;[valid][complete][local]
call;;[check1][check2]
Test Sequence #5
lift
dial
dial;;[valid][complete][local]
connect;;[check1][check2]
ringing
not replace
replace

Test Sequence #6
lift
dial
dial;;[valid][complete][!local]
connect;;[check1][check2]
connect;;[!check1][check3]
connect;;[check1][check3]
connect;;[check1][check3]

Test Sequence #7
lift
dial
dial;;[valid][complete][local]
connect;;[!check2][check3]
connect;;[check2][check3]
connect;;[check2][check3]
connect;;[check2][check3]

Test Sequence #8
lift
dial
dial;;[valid][complete][local]