Black-Box Testing Using Flowgraphs: An Experimental Assessment of Effectiveness and Automation Potential

Stephen H. Edwards
Virginia Tech, Dept. of Computer Science
660 McBryde Hall
Blacksburg, VA 24061-0106 USA
edwards@cs.vt.edu, (540) 231-5723

Abstract

A black-box testing strategy based on Zweben *et al*.'s specification-based test data adequacy criteria is explored. The approach focuses on generating a flowgraph from a component's specification and applying analogues of white-box strategies to it. An experimental assessment of the fault-detecting ability of test sets generated using this approach was performed for three of Zweben's criteria using mutation analysis. By using precondition, postcondition, and invariant checking wrappers around the component under test, fault detection ratios competitive with white-box techniques were achieved. Experience with a prototype test set generator used in the experiment suggests that practical automation may be feasible.

Keywords: automatic testing, object-oriented testing, test data adequacy, formal specification, programming by contract, interface violation detection, self-checking software

1 Introduction

Modular software construction through the assembly of independently developed components is a popular approach in software engineering. At the same time, component-based approaches to software construction highlight the need for detecting failures that arise as a result of miscommunication among components. In component-based software, a component's interface (or specification) is separated from its implementation and is used as a contract between the clients and the implementer(s) of the component [12]. In practice, failures in component-based systems often arise because of semantic interface violations among components—where one party breaks the contract. These errors may not show up until system integration, when they are more expensive to identify and fix. Even worse, internal violations may not be discovered until after deployment. As a result, component-based development increases the need for more thorough testing and for automated techniques that support testing activities.

Testing "to the contract" is at the heart of specification-based testing. This paper describes a strategy for generating black-box test sets for individual software components that was suggested by the work of Zweben, Heym, and Kimmich [19]. The strategy involves generating a flowgraph from a component's specification, and then applying analogues of traditional graph coverage techniques. Section 2 lays out the test set generation approach and describes how it can be applied to object-based or object-oriented software components. Section 3 describes a prototype test set generator based on the described approach that was used to create test sets for evaluation. To compensate for some of the known weaknesses of black-box testing, Section 4 explains how the fault-detecting ability of such tests can be magnified by using interface violation checking wrappers [3]. Section 5 presents an experimental evaluation of the fault-detecting ability of test sets generated using the approach, as assessed through mutation analysis. Section 6 discusses related work, and conclusions are discussed in Section 7. While many interesting and tough research questions remain open, preliminary results suggest practical levels of automation are achievable.

2 The Flowgraph Approach

There are a number of strategies for generating black-box test data from a component's behavioral description [1]. The generation approach taken here is adapted from the work of Zweben *et al.*, who "propose specification-based analogues of control and data flow adequacy criteria for use in testing ADT modules that are specified using a model-based approach" [19]. Their work is directly applicable to both object-based and object-oriented software components.

For the proposed strategy to work, what assumptions are made about components? A component must have a well-defined interface that is clearly distinguishable from its implementation together with a formal description of its intended behavior. The research described here uses formally specified interfaces described in RESOLVE [16], although other model-based specification languages [17] are also applicable. In such a specification, an abstract mathematical model of client-visible state is associated with each type or class, and each operation or method is characterized by pre- and postconditions phrased in terms of such models. More importantly, however, the approach described here gracefully degrades when only semiformal or informal behavioral descriptions are available, as described in Section 3.

2.1 Representing a Specification as a Graph

Zweben *et al.* define a "flowgraph" representation for a software component's specification [19]. Briefly summarized, a component specification's flowgraph is a directed graph where each vertex represents one operation provided by the component, and a directed edge from vertex v_1 to vertex v_2 indicates the possibility that control may flow from v_1 to v_2. Intuitively, an arc from v_1 to v_2 means that there is some legal sequence of operations on the component under consideration where an invocation of v_2 immediately follows an invocation of v_1. For software components that provide initialization or finalization operations (also called constructors or destructors), those are also included as vertices in the graph. For components that represent objects, every possible "object lifetime"—a finite sequence of operations beginning with initialization and ending with finalization—is represented by some path in the graph.

Zweben *et al*. also describe the analogues of the *definition* and *use* of information at a vertex in a flowgraph. All of the parameters for a specific operation, including the implicit "self" or "this" parameter in an object-oriented language, must be considered. A *definition* occurs at a

```
concept Queue_Template
    context
        global context
            facility Standard_Boolean_Facility
        parametric context
            type Item

    interface
        type Queue is modeled by string of math[Item]
            exemplar q
            initialization
                ensures     q = empty_string

        operation Enqueue (
                alters    q : Queue
                consumes  x : Item
            )
            ensures     q = #q * <#x>

        operation Dequeue (
                alters    q : Queue
                produces  x : Item
            )
            requires    q /= empty_string
            ensures     <x> * q = #q

        operation Is_Empty (
                preserves q : Queue
            ) : Boolean
            ensures     Is_Empty  iff  q = empty_string

end Queue_Template
```

Figure 1: A RESOLVE Queue Specification

node for a given parameter if the operation may potentially alter its value. A *use* occurs if the incoming value may affect the behavior of the operation. For each parameter in each operation, definitions and uses may be directly deduced from the postcondition of the operation [19]. Alternatively, definitions and uses may be deduced by examining parameter modes: modes that correspond to "in" or "in/out" data flow identify uses, while "in/out" or "out" data flow identify definitions. For specification languages that do not consider parameter passing modes, care must be taken to ensure correct identification is made; RESOLVE, however, defines parameter modes that are ideal for this purpose [16].

Given such a flowgraph, potential testing strategies become evident [1]. Zweben *et al.* describe natural analogues of white-box control- and data-flow testing strategies adapted to black-box flowgraphs, including node coverage, branch coverage, definition coverage, use coverage, DU-path coverage, and *k*-length path coverage [19]. Unlike program-level testing, where a test case consists of input data for the program, here a test case corresponds to a sequence of operation invocations with associated parameter values. This is similar to the approach used in ASTOOT [2], but with model-based rather than algebraic specifications. Because branches in the graph represent different choices for method calls in a sequence, it is easier to generate test cases that cover any given branch.

2.2 An Example

To ground the discussion, this section illustrates the concept of a flowgraph using a sample component specification. Figure 1 presents the RESOLVE specification for a simple `Queue` component. Notice that the component is generic (a template), and is parameterized by the type of item it contains.

In RESOLVE, every data type implicitly provides three operations: an initialize operation that is implicitly invoked on each object at its point of declaration, a finalize operation that is implicitly invoked on each object when it goes out of scope, and a swap operation (for data movement, instead of assignment) [8]. Thus, the queue component in Figure 1 provides a total of six operations. Figure 2 shows the corresponding flowgraph. Because flowgraphs frequently have large numbers of edges, graphs with more than a handful of vertices are difficult to draw visually and are more effectively represented as matrices, which is why queue was chosen for this example.

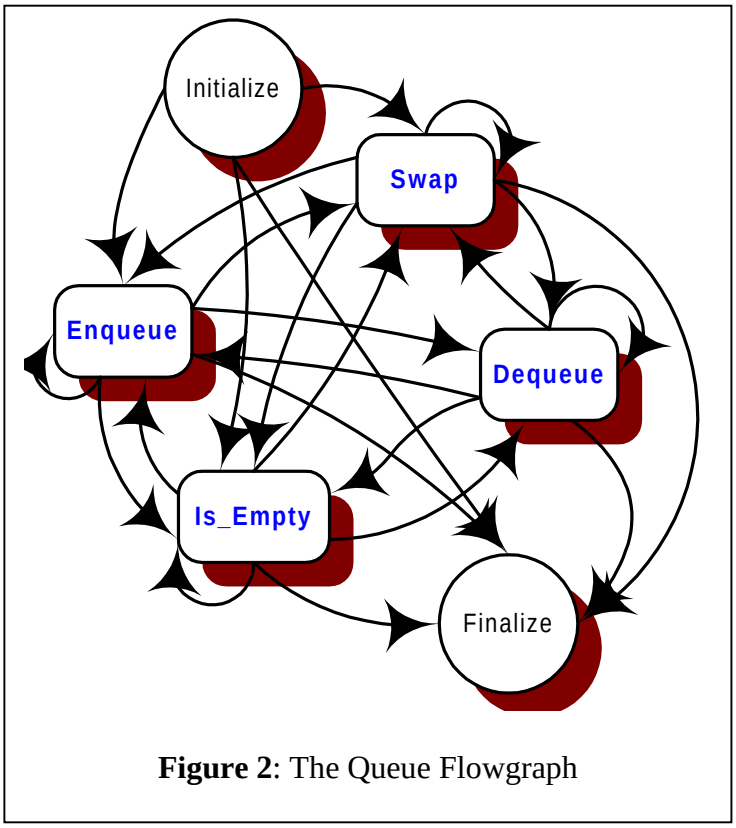

Figure 2: The Queue Flowgraph

2.3 Infeasible Paths

As with white-box approaches, a flowgraph may contain paths that are infeasible. In white-box control flow graphs, an infeasible path may

result from an unreachable node or edge, or it may arise when the conditions necessary to traverse one edge in the path contradict the conditions necessary to traverse some other edge in the path. For specifications, unreachable nodes (i.e., operations that can never be called under any circumstances) are exceedingly rare in practice, but infeasible paths arising from incompatible sequences of operations are common. Each edge (say, from v_1 to v_2) in the flowgraph indicates that there is *some* legitimate object lifetime that includes v_1 followed by v_2; it specifically does not imply that *every* lifetime containing v_1 followed by v_2 is legitimate.

Given the queue example above, the following path exists in the graph: `Initialize` → `Enqueue(q, x)` → `Dequeue(q, x)` → `Dequeue(q, x)` → `Finalize`. This path is infeasible because it violates the precondition of `Dequeue`, a case of the client (rather than the component under test) violating the interface contract. In effect, every legitimate object lifetime is represented by some path in the graph, but not all paths in the graph correspond to legitimate object lifetimes. Such paths are infeasible.

3 Automatically Generating Test Cases

In principle, the steps involved in automatically generating test sets using a flowgraph are clear: select an adequacy criterion, generate the flowgraph for the component under test, systematically enumerate paths in the graph that meet the selected criterion to generate test frames, and finally choose specific parameter value(s) to generate one or more test cases from each frame. As with other black-box test generation strategies, this approach faces several daunting issues in practice. While perfect solutions are not theoretically possible, one may ask whether practical heuristics that provide approximate solutions exist. A prototype test set generator has been implemented with some success, and this section describes the tradeoffs made in its design.

3.1 Choosing Adequacy Criteria

Zweben *et al*. describe a number of specification-based analogues of white-box control and data flow test adequacy criteria [19] that may be used for test set generation: all nodes, all branches, all definitions, all uses, all DU paths, all k-length paths, and all paths. In essence, each such criterion C identifies a set of paths in the flow graph that must be "covered" by any C-adequate test set. The "all paths" criterion is not practical because it includes a potentially infinite set of paths. Three of the remaining criteria were selected for implementation in the initial prototype: all nodes, all definitions, and all uses.

3.2 Generating Flowgraphs

In generating a flowgraph, identifying nodes, definitions, and uses is straightforward. The primary issue is how to correctly and efficiently decide which edges should be included in the graph [19]:

> As is the case for traditional control flow graphs, the question of whether there is a feasible edge from A to B is, in general, undecidable, since otherwise it would be possible to decide whether the conjunction of any two predicate calculus expressions (representing the postcondition of A and the precondition of B, respectively) is satisfiable.

The prototype uses a simplistic, partial solution to this problem coupled with a powerful fallback for difficult-to-decide edges. Experience with RESOLVE-specified components indi-

cates that the vast majority of operations have relatively simple preconditions, although postconditions are usually more complex. As a result, it is practical in many instances to structurally match the precondition for one operation with one clause in the postcondition of another. In effect, this process allows one to identify a subset of the edges that are always feasible (or always infeasible) for every object lifetime. For any remaining edges, The user may select one of three trivial strategies: one may choose to omit all "difficult" edges, decide on an edge-by-edge basis which to include, or include all difficult edges.

Choosing to omit all difficult edges is conservative; it ensures no infeasible edges are included. The cost of this conservatism is exclusion of some feasible edges, and hence exclusion of desirable test cases. Experience with this technique indicates that it rarely leads to desirable results.

Hand-selecting which of the difficult edges are to be included allows one to include only feasible edges and exclude only infeasible ones. The cost of this accuracy is the effort involved in making these choices by hand, together with the risk of human error. For the components used in the experimental analysis in Section 5, this task is relatively easy to perform by hand; only a handful of edges for each component used in the study were not automatically resolved. For larger components or fully automatic operation, the remaining choice is more appropriate.

Choosing to include all difficult edges is liberal, in that it ensures all feasible edges are included. The cost of this liberalism is the inclusion of some infeasible edges, and hence the inclusions of undesirable test cases that force operations to be exercised when their preconditions are false. This is risky, to say the least. Nevertheless, as explained in Section 4, it is possible to *automatically* screen out test cases that exercise infeasible edges. Experience with the prototype suggests that it is much easier to include more test cases than necessary at generation time and automatically weed out infeasible cases later.

3.3 Enumerating Paths

All three of the criteria used in the prototype generate a set of easily identifiable paths for coverage. One test frame can be generated for each such path P (say from v_1 to v_2) by simply finding the shortest path from the `Initialize` node to v_1 (the initialization subpath for P), finding the shortest path from v_2 to the `Finalize` node (the finalization subpath P), and composing the three paths to form the sequence of operations in the test frame. The primary difficulty is again satisfiability: one must not choose an infeasible object lifetime.

The first solution implemented in the prototype was to generate all test frames using this approach and then allow those that are infeasible to be filtered out later, as described in Section 4. In practice, this was less than ideal because of the large number of infeasible test frames produced. Even on simple data structure components where at least one feasible test frame was known to exist for every P generated by this process, the prototype frequently generated infeasible test frames, invariably when P contained a "difficult" edge. In such an instance, the initialization subpath would almost always ensure feasibility of the sequence of operations from `Initialize` through v_1, but some edge in P itself would then be infeasible. The example from Section 2 is typical: `Initialize` $\rightarrow$ `Enqueue(q, x)` $\rightarrow$ `Dequeue(q, x)` $\rightarrow$ `Dequeue(q, x)` $\rightarrow$ `Finalize` is the test frame generated when $v_1 = v_2$ = `Dequeue`.

An alternative heuristic that works much more effectively in practice is to compute the initialization subpath for v_1 (`Initialize` $\rightarrow v_{1,1} \rightarrow \ldots \rightarrow v_{1,m} \rightarrow v_1$), compute the initialization subpath for v_2 (`Initialize` $\rightarrow v_{2,1} \rightarrow \ldots \rightarrow v_{2,n} \rightarrow v_2$), and then use (`Initialize` $\rightarrow v_{1,1} \rightarrow \ldots \rightarrow v_{1,m} \rightarrow v_{2,1} \rightarrow \ldots \rightarrow v_{2,n} \rightarrow v_1$) as the initialization subpath for P, provided that the edge $v_{1,m}$

$\rightarrow v_{2,1}$ exists. Obvious extensions are possible when P contains more than two nodes. In addition, one can modify the method of selecting initialization and finalization subpaths by weighting difficult edges so that paths with a minimum number of difficult edges are selected. Although these heuristics are not guaranteed to produce feasible paths, in practice they worked well at dramatically reducing the number produced in the experiment. Also, note that finalization subpaths rarely cause feasibility problems.

Finally, one should note that this enumeration strategy typically results in naïve test frames—those with the minimal number of operations and minimal number of distinct objects necessary. The result is a larger number of fairly small, directed test cases that use as little additional information as possible.

3.4 Choosing Parameter Values

Finally, one must instantiate the test frames with specific parameter values. This is a difficult problem for scalar parameters, but is even more difficult when testing generic components that may have potentially complex data structures as parameters. As a starting point for the prototype random value selection for scalar parameter types was implemented, which was sufficient for initial experimentation. Later, simple boundary value analysis (BVA) support was added so that scalars with easily describable operational domains could be more effectively supported. A method for supporting user-defined composite types in test cases (including BVA support) was also designed, but there is currently no experience with its application.

In the end, the question of satisfiability for specific parameter values is completely sidestepped—test cases simply are generated and then those that are infeasible are later filtered. Unfortunately, the side-effect of this is more critical: legitimate test frames may be thrown out in practice because the naïve selection of parameter values formed an infeasible test case, even though other feasible parameter value choices may exist. This situation has not yet occurred for the components used in evaluation, but more work in minimizing this risk is necessary.

3.5 Graceful Degredation for Informal Specifications

Although this work presumes that components have formally specified interfaces, it is clear that semi-formal or informal specifications can also be supported. One can generate a flowgraph from as little information as a set of operation signatures that includes parameter modes. One can "liberally" include edges in the flowgraph and then generate test cases in the normal fashion. This runs the risk of generating infeasible test cases, but automatic detection and filtering make this option practical.

4 Automatic Detection of Interface Violations

The effectiveness of the testing strategy described here hinges in great part on automatically detecting interface contract violations for the component under test. Previous work [3] describes a strategy for building wrapper components to perform this function. An interface violation detection wrapper (or "detection wrapper," for short) is a decorator that provides exactly the same client interface as the component it encases. In essence, such a wrapper performs a run-time precondition check for each operation it implements before delegating the call to the "wrapped" component under test. After the underlying component completes its work, the detection wrapper performs a run-time postcondition check on the results. In addition, a detection wrapper may also check component-level invariant properties before and after the delegated call.

Edwards *et al*. describe a novel architecture for the structure of detection wrappers and discuss the potential for partially automated wrapper construction.

Encasing a component under test in a detection wrapper has many significant benefits. A precondition violation identified by the wrapper indicates an invalid (infeasible) test case. This is the primary means for addressing the satisfiability issues raised in Section 3. A postcondition violation identified by the detection wrapper indicates a failure that has occurred within the component. Further, this indication will be raised in the operation or method where the failure occurred, whether or not the failure would be detected by observing the top-level output produced for the test case. Finally, invariant checking ensures that internal faults that manifest themselves via inconsistencies in an object's state will be detected at the point where they occur. Without invariant checking, such faults would require observable differences in output produced by subsequent operations in order to be detected.

Finally, if the component under test is built on top of other components, one should also encase those lower-level components in violation detection wrappers (at least wrappers that check preconditions). This is necessary to spot instances where the component under test violates its client-level obligations in invoking the methods of its collaborators.

Overall, the use of detection wrappers significantly magnifies the fault-detection ability of any testing strategy. That is critically important for specification-based approaches, which in theory cannot guarantee that all reachable statements within the component are executed and thus cannot subsume many white-box adequacy criteria. The use of violation detection wrappers can lead to an automated testing approach that has a greater fault revealing capability than traditional black-box strategies.

5 An Experimental Assessment

As discussed in Section 3, a prototype test set generator for three of Zweben's adequacy criteria has been implemented. The design of the prototype included several tradeoffs, some of which are quite simplistic, that might adversely affect the usefulness of the approach. At the same time, the only empirical test of the fault-detecting ability of test sets following this approach is the original analysis reported by Zweben *et al*. [19]. As a result, an experiment was designed and carried out to examine the fault-detecting ability of test sets generated using this approach—specifically those created using the tradeoffs embodied in the prototype.

5.1 Method

To measure the fault-detecting effectiveness of the test sets under consideration, one can:

- Select a set of components for analysis.
- Use fault injection techniques to generate buggy versions of the components.
- Generate a test set for each of the three criteria, for each of the components.
- Execute each buggy version of each subject component on each test set.
- Check the test outputs (only) to determine which faults were revealed without using violation detection wrappers.
- Check the violation detection wrapper outputs to determine which faults were revealed by precondition, postcondition, and invariant checking.

Four RESOLVE-specified components were selected for this study: a queue, a stack, a one-way list, and a partial map. All are container data structures with implementations ranging

Component	# Methods	Size (NCSLOC)	SLOC/Method	# Mutants
Stack	6	58	9.7	28
Queue	6	60	10.0	27
One-Way List	10	217	21.7	120
Partial Map	7	138	19.7	245
Total	29	473	16.3	420

Table 1: Subject Components in the Experiment

from simple to fairly complex. The queue and stack components both use singly linked chains of dynamically allocated nodes for storage. The primary differences are that the queue uses a sentinel node at the beginning of its chain and maintains pointers to both ends of the chain, while the stack maintains a single pointer and does not use a sentinel node. The one-way list also uses a singly linked chain of nodes with a sentinel node at the head of the chain. Internally, it also maintains a pointer to the last node in the chain together with a pointer to represent the current list position. The partial map is the most complex data structure in the set. It is essentially a dictionary that stores domain/range pairs. Internally its implementation uses a fixed-size hash table with one-way lists for buckets. The ordering of elements in an individual bucket is unimportant; buckets are searched linearly.

Although these components are all relatively small, the goal is to support the testing of software components, including object-oriented classes. The components selected here are more representative of classes that maintain complex internal state than they are of complete programs, which is appropriate. In addition, the size of the subject components was important in supporting a comprehensive fault injection strategy without incurring undue cost.

The fault injection strategy chosen was based on *expression-selective mutation testing* [13]. This version of mutation testing uses only five mutation operators: ABS, AOR, LCR, ROR, and UOI [10]; it dramatically reduces the number of mutants generated, but has been experimentally shown to achieve almost full mutation coverage [13]. Each mutant was generated by applying one mutation operator at a unique location in one of the methods supported by a component. The number of generated mutants was small enough to admit hand-identification of equivalent mutants. All remaining mutants were guaranteed to differ from the original program on some legitimate object lifetime in a manner observable through the parameter values returned by at least one method supported by the component. Table 1 summarizes the characteristics of the subject components and number of faulty versions generated.

5.2 Results

Because the goal was to assess test sets generated using the specific heuristics described in this paper rather than the more general goal of assessing the adequacy criteria themselves, the experiment was limited to test sets produced by the prototype. The prototype uses a deterministic process to generate test frames, however, so producing multiple test sets for the same component and criterion would only result in random variations in parameter values, rather than any substantial variations in test case structure. Since all of the subject components are data structures that are relatively insensitive to the values of the items they contain, such superficial variations in generated test sets were not explored. As a result, a total of 12 test sets were generated,

Adequacy Criterion	Subject	# Cases	Observed Failures	%	Detected Violations	%	Infeasible Cases
All Nodes	Stack	5	16	57.1%	22	78.6%	0
	Queue	5	15	55.6%	21	77.8%	0
	One-Way List	10	37	30.8%	89	74.2%	0
	Partial Map	7	118	48.2%	153	62.4%	0
	Total	27	186	44.3%	285	67.9%	0
All Definitions	Stack	6	6	21.4%	16	57.1%	0
	Queue	6	5	18.5%	15	55.6%	0
	One-Way List	11	47	39.2%	83	69.2%	0
	Partial Map	6	92	37.6%	132	53.9%	0
	Total	29	150	35.7%	246	58.6%	0
All Uses	Stack	32	26	92.9%	28	100.0%	3
	Queue	32	25	92.6%	27	100.0%	3
	One-Way List	126	106	88.3%	120	100.0%	10
	Partial Map	61	185	75.5%	214	87.3%	11
	Total	251	342	81.4%	389	92.6%	27

Table 2: Expression-Selective Mutation Scores of Test Sets

one for each of the chosen adequacy criteria for each of the subject components. Inclusion of "difficult" edges in the corresponding flowgraphs was decided by hand on a per edge basis.

For each subject component, the three corresponding test sets were run against every faulty version. Table 2 summarizes the results. "Observed Failures" indicates the number of mutants killed by the corresponding test case based solely on observable output (without considering violation detection wrapper checks). "Detected Violations" indicates the number of mutants killed solely by using the invariant and postcondition checking provided by the subject's detection wrapper. In theory, any failure identifiable from observable output will also be detected by the component's postcondition checking wrapper if the wrapper is implemented correctly; the data collected were consistent with this expectation, so every "Observed Failure" was also a "Detected Violation." The rightmost column lists the number of infeasible test cases produced by the prototype in each test set.

5.3 Discussion

On the subject components, it is clear that "all definitions" reveals the fewest faults of the three criteria studied, while "all uses" reveals the most, which is no surprise. More notable is the magnification of fault-detecting power supplied by the use of violation detection wrappers. In all cases, the use of detection wrappers significantly increased the number of mutants killed. Further, the increase was more dramatic with weaker test sets. In the extreme case of the "all nodes" test set for one-way list, its fault-detection ability was doubled.

Also surprising is the fact that for three of the four subjects, the "all uses" test sets achieved a 100% fault detection rate with the use of detection wrappers. This unusual result bears some interpretation. It appears to be the result of two interacting factors. First, the invariant checking performed by each detection wrapper is quite extensive (partly because formal statements of representation invariants were available for the subjects), providing the magnification effect discussed above. Second, the simpler subject components involved significantly

fewer complex logic conditions and nested control constructs. As a result, the "all nodes" and "all uses" test sets for the stack, queue, and one-way list components actually achieved 100% white-box statement-level coverage of all statements where mutation operators were applied. This fact was later confirmed via code instrumentation. Presumably this is atypical of most components, so the 100% fault detection results for "all uses" should not be unduly generalized. Nevertheless, the fact that a relatively weak adequacy criterion could lead to such effective fault revelation is a promising sign for this technique.

The only other empirical study of specification-based testing based on these criteria is reported by Zweben *et al*. [19]. That work reports a small study on 25 versions of a two-way list component written by undergraduate students. While small in scale, it does give some indication of the fault-detecting ability of the various specification-based control- and data flow criteria on defects that occurred in real components. In that study, "all nodes" revealed 6 out of 10 defects, "all definitions" revealed 6 out of 10, and "all uses" revealed 8 out of 10, all of which are comparable to the "Observed Failures" results here.

Although the results of this experiment are promising, there are also important threats to the validity of any conclusions drawn from it. The subjects were limited in size for practical reasons. Although well-designed classes in typical OO designs are often similar in size, it is not clear how representative the subjects are in size or logic complexity. Also, the question of how well mutation-based fault injection models real-world faults is relevant, and has implications for any interpretation of the results. With respect to the adequacy criteria themselves, this experiment only aims to assess test sets generated using the strategy described in this paper, rather than aspiring to a more sweeping assessment of the fault-detecting ability of the entire class of test sets meeting a criterion.

6 Related Work

The test set generation approach described here has been incorporated into an end-to-end test automation strategy that also includes generation of component test drivers and partial to full automation of violation detection wrappers [4]. The most relevant related projects in automated generation of specification-based test sets are DAISTS [7] and ASTOOT [2]. One key difference with the current work is that model-based specifications are used while DAISTS and ASTOOT are based on algebraic specifications. Algebraic specifications often encourage the use of function-only operations and may suppress any explicit view of the content stored in an object. Model-based specifications instead focus on abstract modeling of that content, direct support for state-modifying methods, and direct support for operations with relational behavior. Another notable difference is the use of violation detection wrappers in this approach, which provides increased fault-detecting ability.

Other specification-based testing or test generation approaches focus on finite state machine (FSM) models of classes or programs [1, 9, 14, 15]. The work of Hoffman, Strooper, and their colleagues is of particular relevance because it also uses model-based specifications. An FSM model typically contains a subset of the states and transitions supported by the actual component under consideration, and may be developed by identifying equivalence classes of states that behave similarly. Test coverage is gauged against the states and transitions of the model. The work described here, by contrast, does not involve collapsing the state space of the component under test, or even directly modeling it. As a result, it gracefully degrades when only semi- or informal specifications are available. In addition, the use of violation detection wrappers sets the current work apart from FSM approaches.

Other work on test data adequacy, including both specification-based and white-box criteria, is surveyed in detail by Zhu [18]. The focus of the current work is to develop and assess practical test set generation strategies based on existing criteria, rather than describing new criteria. Similarly, Edwards *et al*. [3] provide a more detailed discussion of interface violation detection wrappers and the work related to them, including alternative approaches to run-time postcondition checking.

There is a large body of work on experimentally assessing the fault-detecting ability of various testing strategies or adequacy criteria [5, 6, 11]. Frankl's work on statistically characterizing the effectiveness of adequacy criteria is notable. Whereas her work focuses on statistically assessing the effectiveness of criteria by looking at large numbers of test sets, the work here instead aims at assessing test sets created using a specific strategy.

7 Conclusions

This article describes a specification-based test set generation strategy based on Zweben *et al*.'s specification-based test data adequacy criteria. The strategy involves generating a flowgraph from a component's specification, and then applying analogues of white-box strategies to the graph. Although there are a number of very difficult issues related to satisfiability involved in generating test data, a prototype test set generator was implemented using specific design tradeoffs to overcome these obstacles. An experimental assessment of fault-detecting ability based on expression-selective mutation analysis provided very promising results. By using precondition, postcondition, and invariant checking wrappers around the component under test, fault detection ratios competitive with white-box techniques were achieved. The results of the experiment, together with experiences with the generator, indicate that there is the potential for practical automation of this strategy.

References

1. Beizer B. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* Wiley: New York, 1995.
2. Doong R-K, Frankl PG. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Software Engineering Methodology*, 1994, **3**(2): 101-130.
3. Edwards S, Shakir G, Sitaraman M, Weide BW, Hollingsworth J. A framework for detecting interface violations in component-based software. *Proc. 5th Int'l Conf. Software Reuse*, IEEE CS Press: Los Alamitos, CA,1998, pp. 46-55.
4. Edwards SH. A framework for practical, automated black-box testing of component-based software. *Proc. 1st Int'l Workshop on Automated Program Analysis, Testing and Verification*, June 2000, pp. 106-114.
5. Frankl PG, Weiss SN. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Software Engineering*, Aug. 1993, **19**(8): 774-787.
6. Frankl PG, Weiss SN, Hu C. All-uses versus mutation testing: An experimental comparison of effectiveness. *J. Systems and Software*, Sept. 1997, **38**(3): 235-253.
7. Gannon JD, McMullin PR, Hamlet R. Data-abstraction implementation, specification, and testing. *ACM Trans. Programming Languages and Systems*, July 1981, **3**(3): 211-223.
8. Harms DE, Weide BW. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Software Engineering*, May 1991, **17**(5): 424-435.

9. Hoffman D, Strooper P. The test-graphs metholodogy: Automated testing of classes. *J. Object-Oriented Programming*, Nov./Dec. 1995, **8**(7): 35-41.
10. King KN, Offutt J. A Fortran language system for mutation-based software testing. *Software Practice and Experience*, July 1991, **21**(7): 686-718.
11. Mathur AP, Wong WE. Comparing the fault detection effectiveness of mutation and data flow testing: An empirical study. Tech. Report SERC-TR-146-P, Software Engineering Research Center, Dec. 1993.
12. Meyer B. *Object-Oriented Software Construction, 2nd Edition.* Prentice Hall PTR: Upper Saddle River, New Jersey, 1997.
13. Offutt J, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Trans. Software Engineering Methodology*, April 1996, **5**(2): 99-118.
14. Offutt J, Abdurazik A. Generating tests from UML specifications. *2nd Int'l Conf. Unified Modeling Language (UML99)*, Fort Collins, CO, Oct. 1999.
15. Offutt J, Xiong Y, Liu S. Criteria for generating specification-based tests. *5th IEEE Int'l Conf. Engineering of Complex Computer Systems (ICECCS '99)*, Las Vegas, NV, Oct. 1999.
16. Sitaraman M, Weide BW, eds. Component-based software engineering using RESOLVE. *ACM SIGSOFT Software Enigineering Notes*, 1994, **19**(4): 21-67.
17. Wing JM. A specifier's introduction to formal methods. *IEEE Computer*, Sept. 1990, **29**(9): 8-24.
18. Zhu H, Hall PAV, May JHR. Software unit test coverage and adequacy. *ACM Computing Surveys*, Dec. 1997, **29**(4): 366-427.
19. Zweben S, Heym W, Kimmich J. Systematic testing of data abstractions based on software specifications. *J. Software Testing, Verification and Reliability*, 1992, **1**(4): 39-55.