A Framework for Practical, Automated Black-Box Testing of Component-Based Software

Stephen H. Edwards
Virginia Tech, Dept. of Computer Science
660 McBryde Hall
Blacksburg, VA 24061-0106 USA
edwards@cs.vt.edu, +1 540 231 5723

Abstract

This paper outlines a general strategy for automated black-box testing of software components that includes: automatic generation of component test drivers, automatic generation of black-box test data, and automatic or semi-automatic generation of component wrappers that serve as test oracles. This research in progress unifies several threads of testing research, and preliminary work indicates that practical levels of testing automation are possible.

Keywords: test drivers, test oracles, self-checking software, test adequacy, integration testing, modular construction, and built-in test

1 Introduction

The move toward component-based software development offers many promises for improved productivity and quality, but it also highlights the need for effective methods of testing reusable software parts. Indeed, components that are independently developed or commercially purchased underscore the need to detect errors well before system integration time. As components increase in complexity, this need only becomes more urgent. Adopting a systematic approach to testing components and automating as much of the approach as possible is one solution. To this end, this paper outlines a general strategy for automated black-box testing of software components. The strategy involves a three-pronged attack, covering automatic generation of component test drivers, automatic generation of test data, and automatic or semi-automatic generation of wrappers serving the role of test oracles. In combination, these thrusts build toward a complete, “end to end” component testing strategy that is almost entirely automated. This work unifies several threads of testing research into a coherent whole. While many interesting and tough research questions remain open, preliminary results suggest practical levels of automation are achievable for components that include formal behavioral descriptions.

Section 2 describes the assumptions about components that are necessary for the approach to work, and presents an example component specification satisfying these assumptions. Section 3 discusses a critical piece of the strategy presented here: the use of pre- and postcondition checking wrappers around the component under test. Building on this foundation, Section 4 lays out the vision for an automated testing framework. Section 5 briefly discusses related work, followed by open research issues and future directions in Section 6.

2 An Example Component: One-Way List

The strategy proposed here builds on one key requirement: a component must have a clear description of its intended behavior. Such a description is important when automating the tasks of generating test data or checking test results. The initial requirement for the research described here is that a component must have a formally specified interface described in a model-based specification language. RESOLVE [18] has been selected as the specification language for this research, although other model-based specification languages [20] are also applicable. The choice of specification language was made for two pragmatic reasons: the researchers involved were familiar with the language, and using it provides a natural collaboration path for fielding tools. Researchers at The Ohio State University and at West Virginia University are collaborating on a Software Composition Workbench based on RESOLVE technology that is an ideal environment in which to evaluate and apply the testing tools described in this paper.

Although the initial research requirement is that all components have formally specified interfaces, the tools making up the approach do provide graceful fallback positions if only semi-formal or informal component behavioral descriptions are available. Informal descriptions require more human intervention in the process, however, since there is no easy way to automatically extract behavioral requirements. The end result is a strategy that can still be applied, even without any formal behavioral descriptions, but at the cost of reduced automation and greater programmer intervention.

In general, a software component may be as simple as an individual class or module, or as sophisticated as a Java Bean or COM object. In the interests of simplicity, this paper will present examples in terms of object-oriented classes, although the techniques are easily generalized for much larger-grained components.

```
concept One_Way_List
    context
        global context
            facility Standard_Boolean_Facility
        parametric context
            type Item
    interface

        type List is modeled by (
             left  : string of math[Item],
             right : string of math[Item]
           )
            exemplar s
            initialization
                ensures     s = (empty_string, empty_string)

        operation Move_To_Start (alters s : List)
            ensures     s = (empty_string, #s.left * #s.right)

        operation Move_To_Finish (alters s : List)
            ensures     s = (#s.left * #s.right, empty_string)

        operation Advance (alters s : List)
            requires    s.right /= empty_string
            ensures     there exists x : Item
                            (s.left = #s.left * < x >  and  #s.right = < x > * s.right)

        operation Add_Right (alters s : List,  consumes  x : Item)
            ensures     s = (#s.left, < #x > * #s.right)

        operation Remove_Right (alters s : List, produces x : Item)
            requires    s.right /= empty_string
            ensures     s.left = #s.left  and  #s.right = < x > * s.right

        operation Swap_Rights (alters s1 : List, alters s2 : List)
            ensures     s1.left = #s1.left  and  s1.right = #s2.right  and
                        s2.left = #s2.left  and  s2.right = #s1.right

        operation At_Start (preserves s : List) : Boolean
            ensures     At_Start  iff  s.left = empty_string

        operation At_Finish (preserves s : List) : Boolean
            ensures     At_Finish  iff  s.right = empty_string

end One_Way_List
```

Figure 1—A RESOLVE Specification for One-Way List

To ground the discussion of formally specified components in this paper, Figure 1 presents the RESOLVE specification of a one-way list component that was originally described by Sitaraman *et al*. [17]. This generic component is parameterized by the type of `item` it will contain. A one-way list is an ordered sequence of items, all of the same type. One may move forward in the sequence, accessing individual elements in turn, or jump to either end of the sequence. Zweben presents a similar component that supports bi-directional movement [22]. A one-way list may be implemented as a singly linked chain of dynamically allocated nodes, as a dynamically allocated array, or by building on other components like a stack, a queue, or a vec-

vector, among other alternatives. Figure 2 sketches an equivalent abstract base class template in C++.

The mathematical model of the one-way list shown in Figure 1 is a pair of mathematical strings (finite sequences). There is no explicit notion of a "current position" or "cursor." Instead, the current location is implicit in the fact that the string is partitioned into left and right segments. Intuitively, items to the "left" are those that are "behind" the current location (closer to the front of the sequence), while those to the "right" are in front (toward the rear). The preconditions (`requires` clauses) and postconditions (`ensures` clauses) of each operation supported by the component are described in terms of this mathematical model. In postconditions, the hash symbol (#) is used to refer to the incoming value of a parameter, rather than its outgoing value.

```
template <class Item>
class One_Way_List {
public:
    One_Way_List ();
    virtual ~One_Way_List ();
    // ...
    virtual void Remove_Right (Item& x) = 0;
    // ... other details elided for space
};
```

Figure 2—A C++ Interface for One-Way List

3 The Central Focus: Built-In Test Capabilities

The cornerstone of the automated testing framework is a micro-architecture for providing built-in test (BIT) support in software components. This architecture builds on current research in systematically detecting interface violations in component-based software [5]. In essence, each software component provides a simple "hook" interface (with no run-time overhead) that can be used in adorning the component with sophisticated BIT capabilities. Figure 3 illustrates this idea. Sophisticated "decorator" components (wrappers) that provide a number of self-checking and self-testing features can then be used to encase the underlying component.

The innovative properties of this strategy are:

- BIT wrappers are completely transparent to client and component code.
- BIT wrappers can be inserted or removed without changing client code (only a declaration need be modified). This capability does not require a preprocessor, and can be used in most current languages.
- When BIT support is removed, there is no run-time cost to the underlying component.
- Both internal and external assertions about a component's behavior can be checked.
- Precondition, postcondition, and abstract invariant checks can be written in terms of the component's abstract mathematical model [5], rather than directly in terms of the component's internal representation structure.
- Checking code is completely separated from the underlying component.
- Violations are detected when they occur and before they can propagate to other components; the source of the violation can be reported down to the specific method/operation responsible.
- Routine aspects of the BIT wrappers can be automatically generated.

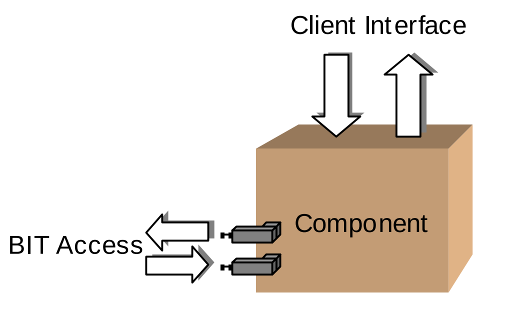

Figure 3–Component Provides "Hooks" for BIT Infrastructure

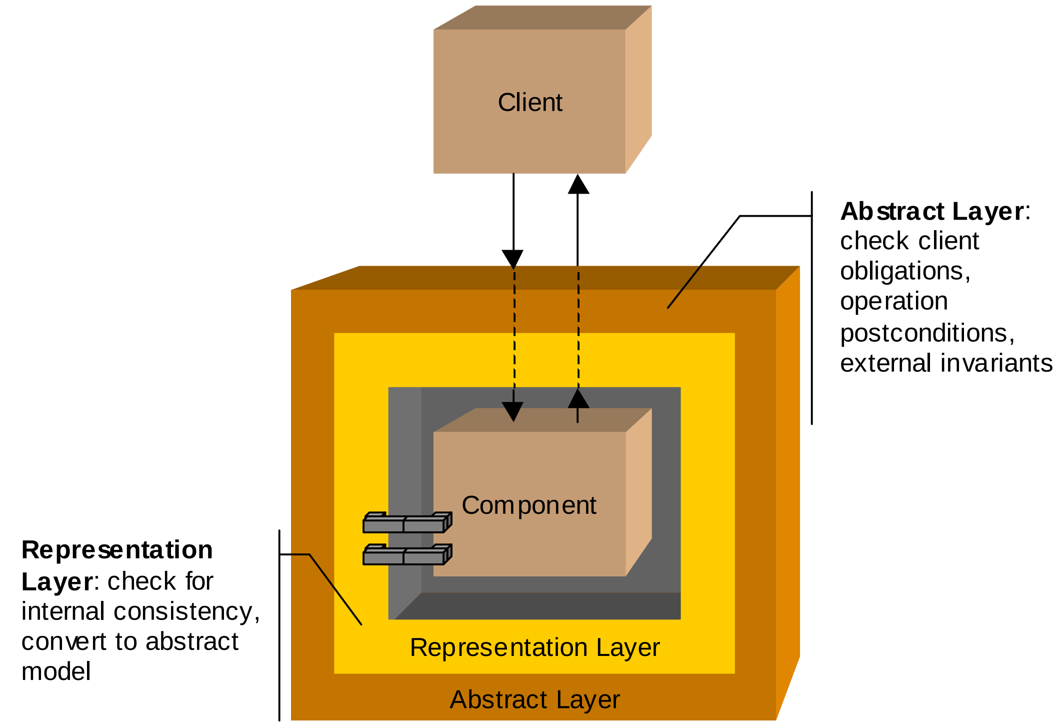
Client
Abstract Layer: check client obligations, operation postconditions, external invariants
Component
Representation Layer: check for internal consistency, convert to abstract model
Representation Layer
Abstract Layer

Figure 4–A Wrapper Surrounds the Component, Implementing All Necessary Testing Functions

- The approach works well with formally specified components, but does not require formal specification.
- The approach provides full observability of a component's internal state without breaking encapsulation for clients.
- Actions taken in response to detected violations are separated from the BIT wrapper code.

Figure 4 illustrates a component encased in a two-ply BIT wrapper. The inner layer of the wrapper is responsible for directly and safely accessing the component's internals, performing internal consistency checks, and then converting the internal state information into a program-manipulable model of the component's abstract state [5]. The outer layer is responsible for using this model to check that clients uphold their obligations in using the underlying component, to check that the component maintains any invariant properties it advertises, and to double-check the results of each operation to the extent desired for self-testing purposes. Client code accesses the component just as if it were unadorned.

```
template <class Item, class One_Way_List_Base>
class OWL_BIT_Wrapper :
    public One_Way_List <Item> {
public:
    OWL_BIT_Wrapper ();
    virtual ~OWL_BIT_Wrapper ();
    // ...
    virtual void Remove_Right (Item& x);
    // ... other details elided for space
private:
    One_Way_List_Base uut; // wrapped list
};
```

Figure 5—A C++ One-Way List BIT Wrapper

Figure 5 outlines the class declaration for a BIT wrapper in the context of the one-way list example. Although describing the implementation of such a wrapper is beyond the scope of this article, full details appear elsewhere [5]. Figure 6 provides pseudocode for the checking process used in the wrapper by outlining one operation.

```
template <class Item, class One_Way_List_Base>
void OWL_BIT_Wrapper<Item, One_Way_List_Base>::
   Remove_Right (Item& x)
{
   // Store abstract value information
   Model_Of_List incoming_self, incoming_self;
   Model_Of_Item outgoing_x,    outgoing_x;

   // pseudocode:
   Check internal invariant properties for uut
   Project abstract view of uut's state in incoming_self
   Ask x to project its abstract value into incoming_x
   Check abstract invariant properties for incoming_self
   Check precondition on incoming values

   uut.Remove_Right (x);  // invoke wrapped unit

   // pseudocode:
   Check internal invariant properties for uut
   Project abstract view of uut's state in outgoing_self
   Ask x to project its abstract value into outgoing_x
   Check abstract invariant properties for outgoing_self
   Check postcondition on incoming/outgoing values
}
```

Figure 6—Implementing a BIT Wrapper Method

Two issues that are critical in the design of a BIT wrapper are interference and overhead. First, one must be sure that the wrapper does not interfere with the behavior of the underlying component, so that only that component's inherent behavior will affect the results of tests. A BIT wrapper guarantees that it makes no modifications to the wrapped object, and instead performs virtually all actions on a copy of the object's abstract state [5]. This leads to the second issue: the overhead imposed by a BIT wrapper. The goals for BIT wrappers have led to a design that encourages more comprehensive (and expensive) checks that are easily enabled or disabled through simple declaration changes. The expectation is that BIT wrappers will be used during functional testing and integration, but will be removed for non-functional tests (time or space performance, for example) or for delivery.

The BIT strategy is designed to provide maximal support during unit testing, debugging, and integration testing. By outfitting a component with a BIT wrapper during unit testing, much more thorough testing can be achieved with the ad hoc strategies most developers employ. For every test case executed, a large number of internal consistency checks are performed, any one of which has the potential of revealing errors. Since these checks are automatically performed for any and all operations executed by the component in each test case, they have the effect of multiplying the tester's ability to detect errors. When errors are found, the full visibility of internal state provided by the BIT strategy is helpful during debugging. In particular, the strategy provides the programmer with additional capabilities for both input and output of internal state information, as well as the ability to modify internal state information for debugging purposes. None of these capabilities require any additional design or coding time from the developer, beyond the inclusion of the original BIT hooks in the underlying component. Finally, during integration testing, BIT wrappers can provide firewalls between components for incremental integration. As new units are added to the system, the wrappers will detect any unforeseen interactions. This strategy supports bottom-up, top-down, and hybrid incremental integration strategies.

4 The Vision: An Automated Testing Framework

The BIT infrastructure provides a natural mechanism for supporting semi- or fully automatic testing. Simply put, the framework described here rests on three legs:

- Automatic (or semi-automatic) generation of a component's BIT wrapper.
- Automatic generation of a component's test driver.
- Automatic (or semi-automatic) generation of test cases for the component.

All three generation strategies rely on the same information: a complete behavioral description of the component's interface contract. By combining these generation strategies, it is possible to create a test driver, a test suite, and a BIT wrapper directly from a component's specification. If the BIT wrapper also provides comprehensive checks on the postconditions of all exported operations—in effect, acting as a test oracle—then the combination will produce a highly automated testing and debugging capability, as outlined in Figure 7.

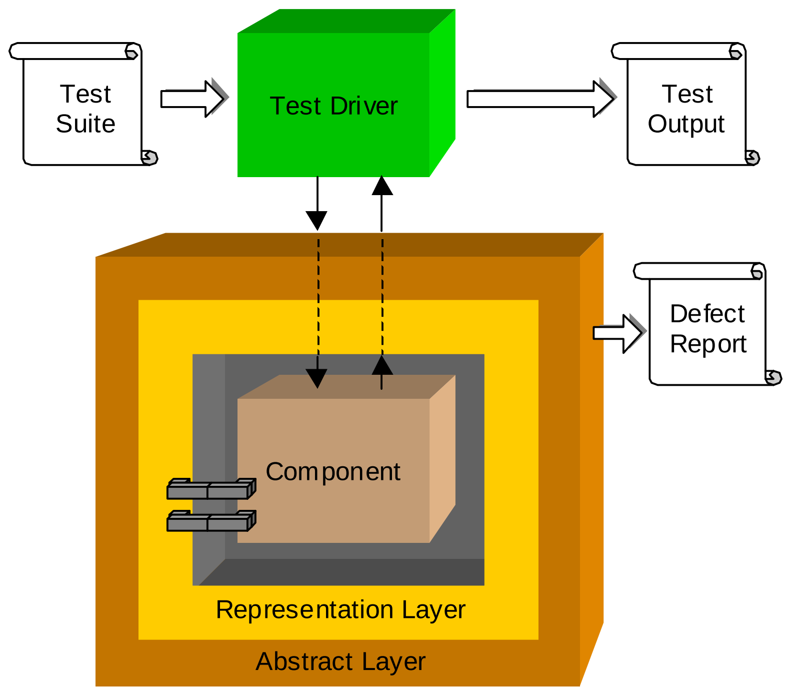

Figure 7–General Testing Setup

4.1 Generating BIT Wrappers

A generator has been designed and implemented to process RESOLVE-style component specifications and C++ template interfaces to generate BIT wrappers [16]. The underlying principles for creating such wrappers are independent of any particular specification technique or implementation language, and they can be readily extended to other languages [5].

The external interface of a BIT wrapper is identical to that of the corresponding base component. Semantically, they differ in how they behave when either the pre- or postcondition of some operation is violated. In particular, where a regular component guarantees nothing if an operation is invoked under conditions violating its precondition, a BIT wrapper instead guarantees it will perform a specific notification action. A BIT wrapper that only checks for precondition violations is called a one-way checking wrapper.

Similarly, a two-way checking wrapper guarantees to:

1. Carry out its precondition notification action if the precondition does not hold, or
2. Establish that the postcondition is true upon operation completion, or
3. Carry out its postcondition notification action if the postcondition does not hold.

Both one-way and two-way checking wrappers are extremely useful. One-way wrappers correspond with the traditional notion of a "defensive shell" that protects a component from errant clients. Two-way wrappers, on the other hand, are more akin to "self-checking" or "self-verifying" components that confirm their own work as well as spotting erroneous client behavior.

While constructing BIT wrappers is a straightforward process, it raises the question of how one can automatically generate pre- and postcondition checks. For most components, checking each precondition is straightforward and can thus be automated. By using the model conversion approach described in [5], many precondition and postcondition assertions can be converted to code by a simple transliteration process. For example, complete pre- and postcondition checks can be automatically generated for the one-way list specification in Figure 1.

However, some assertions are non-trivial. For example, code for checking assertions containing quantifiers cannot be generated mechanically [2]. To provide greater support for automated BIT wrapper construction in the face of such difficulties, three possibilities are being explored: semi-automatic generation, dynamic assertion verification, and "armored" components that use reference implementations.

4.1.1 Semi-Automatic Generation

It is possible to automatically generate checking code for many preconditions as well as for many clauses in postconditions. One possible approach to solving this problem is to automatically generate everything that is appropriate, and allow a human to provide the code for those checks that cannot be automated. Experience with the prototype wrapper generator indicates it is a simple process to separate the human-contributed checks from all of the other infrastructure code necessary to support a BIT wrapper. Further, the person creating the checks will write them in abstract client-level terms—i.e., the mathematical model of the component's state—instead of in terms of the implementation of the component under test [5].

4.1.2 Dynamic Assertion Verification

Another alternative uses current generation verification tools. While current verification tools often have trouble with complex quantified assertions that arise during static formal verification, the simpler assertions that arise at run-time in a BIT wrapper, where all variables have specific values, are more amenable to existing proof tools. It is possible to automatically generate a complete BIT wrapper that relies on a verification/proof engine for assertion checking with specific parameter values at run-time, an approach termed "dynamic verification" [19].

4.1.3 "Armored" Components Using Reference Implementations

If a reference implementation for a component (even an inefficient one) exists, it is possible to automatically generate a BIT wrapper that executes both the unit under test and the reference implementation and then uses the results of the reference implementation to judge the correctness of the unit under test. This form of testing, also called *back-to-back testing*, is sometimes used on a larger scale but for the same purpose in conformance or compliance testing. If one wishes to ensure that a candidate implementation meets some published API standard, for example, a trusted reference implementation can be used as a judge in assessing the candidate implementation over a standard test suite. In effect, the same approach can be used in-the-small and in an automated fashion within a BIT wrapper.

Such an approach invites intriguing enhancements, since it is possible to recover from internal errors; the wrapper, which stands between the client and the two implementations, can selectively pass on the reference implementation results when the unit under test fails. Further, the BIT infrastructure can easily be extended to allow the “good” data produced by the reference implementation to be used to force recovery on the unit under test [5]. This leads to a defensive wrapper that is close to bulletproof.

4.2 Generating Test Drivers

Compared to the difficulties involved in generating BIT wrappers, generating test drivers is a simpler problem. The research described here is based on an interpreter model for test drivers: a test driver can be viewed as a command interpreter that reads in test cases and translates them into actions on the component under test. From this point of view, it is straightforward to parse a component's interface definition, identify its operations, and construct an interpreter. All filtering of invalid operation requests is handled by the BIT wrapper encasing the component under test, as is run-time checking of produced output. The major weaknesses of this approach are in effectively handling components that rely on inversion of control or that have a substantial human interaction component.

A test driver generator based on this strategy has been designed and is currently being implemented. RESOLVE/C++ serves as the underlying implementation language for components in this preliminary work [18], so a subset of C++ was adopted as a test case definition language. Figure 8 shows a sample test case for the one-way list component. In this test case, `List` is the test driver’s name for the unit under test—an `OWL_BIT_Wrapper` surrounding a concrete class implementing the `One_Way_List` specification and instantiated to contain integer elements.

The architecture for the interpreter/test driver uses the envelope and letter paradigm for handling internal values, and uses an exemplar-based dispatching strategy for handling operations on user-defined objects [3]. As a result, the core interpreter engine does not directly refer to the component under test or any of its methods. This means that support for any unit under test can be added without requiring any changes to or recompilation of the interpreter engine itself. Instead, the driver generator creates a “glue” source file that, when compiled and then linked with the existing interpreter object files, produces a custom driver for the component under test. Preliminary experience with this approach indicates that an interpreter provides significant timesavings over direct compilation of test cases when large test sets are used.

4.3 Generating Test Data

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 based on flowgraphs and is described more thoroughly elsewhere [6]. It is adapted from black-box test adequacy criteria described by Zweben et al. This black-box test adequacy work describes how one can construct a flow graph from a behavioral specification. This directed graph has a single entry, representing object creation, and a single exit, representing object destruction. Every “object lifetime”—composed of some legal sequence of operations applied to a given object—is represented as some (possibly cyclic) path through the graph.

```
{
   List l;
   Integer x;
   x = 43751;
   l.Add_Right (x);
   l.Remove_Right (x);

   cout << "output => " << l
        << ' ' << x << endl;
}
```

Figure 8—A One-Way List Test Case

Given such a flow graph, possible testing strategies become evident [1]. Zweben et al. describe natural analogues of white-box control- and data-flow testing strategies adapted to black-box flow graphs, including node coverage, branch coverage, all definition coverage, all use coverage, all DU-path coverage, and all k-length path coverage. Further, because branches in the graph represent different choices for method calls in a sequence, instead of logical control-flow decisions, it is easier to generate test cases that exercise all branches.

As with other black-box test generation strategies, this approach faces two open issues: how to correctly and efficiently decide which edges should be included in a graph, and how to address the problem of satisfiability in choosing test data values to be used in individual test cases. While perfect solutions to these problems are not computable, practical heuristics that provide approximate solutions are available [6]. When combined with a BIT wrapper surrounding the component under test, invalid test cases can be automatically screened and removed, allowing overly optimistic heuristics to be used in practice. Further, the internal checks performed by BIT wrappers have the possibility of revealing defects that are not directly observable from the output produced by operations. This property can lead to an automated testing approach that has a greater defect revealing capability than traditional black-box strategies [6].

A preliminary evaluation of the effectiveness of this approach provided encouraging evidence for its feasibility [6]. An experiment was conducted on four RESOLVE-specified components (a stack, queue, one-way list, and partial map), where defects were seeded using a mutation-based approach [21]. Faults were injected using *expression-selective mutation* [12]. This version of mutation testing uses only five mutation operators: ABS, AOR, LCR, ROR, and UOI [9]; it significantly reduces the number of mutants generated, but has been experimentally shown to achieve nearly complete mutation coverage [12]. All equivalent mutants were identified by hand and removed from the collection to ensure that the mutants used in the study were guaranteed to differ from the original on some legitimate object lifetime in a manner observable through the parameter values returned by at least one of the component's methods.

Test sets for each of three adequacy criteria were used in the evaluation. The results of the experiment indicate that a black-box analog to the "all uses" criteria was extremely effective in identifying faults, detecting 100% of the faults seeded in three of the four components, and 87.3% of the faults seeded in the most complicated component in the experiment (the partial map).

Further, the experiment also used hand-written BIT wrappers and separately tabulated the number of defects detectable by observing a component's output versus those detected by the internal checks performed by the BIT wrapper. Figure 9 provides a graphical summary of the percentage of mutants killed for each adequacy criterion and component, together with the increased detection rate provided by BIT wrappers.

The use of two-way checking BIT wrappers provided an improvement in defect revealing capability in every case where they were used, ranging from 8%–200%

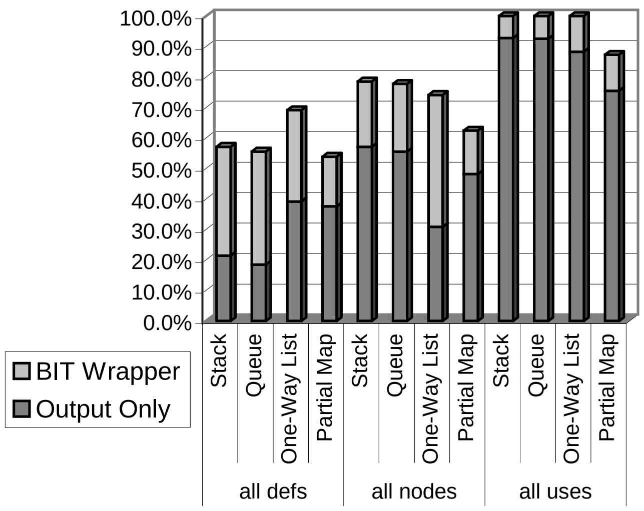

Figure 9—Defect Detection Rates

more mutants killed. The greatest improvement was seen in the weakest test sets; for example, the all definitions test set for the queue component only revealed 18.5% of defects by examining test output alone, but this rate increased to 55.6% with a BIT wrapper. Further, the test sets that achieved 100% detection of seeded defects required the use of BIT wrappers to do so, since not all defects were detected by component output.

The magnification effect that BIT wrappers provide for revealing defects is interesting to explore. Because all mutants used in the study were observably different from the original for some object lifetime, any seeded defect could have been uncovered solely by observing the output of some test case. However, not all test cases that cause a defect to be executed necessarily produce observably different behavior. As an example, suppose the `One_Way_List` component described in Section 2 is implemented as a singly linked chain of dynamically allocated nodes, stored together with an integer recording the length of the list. A number of seeded defects for such a component lead to internal corruption of encapsulated data during one object method call that can then be observed on subsequent method calls. For the `One_Way_List`, the stored length may become incorrect, the chain of nodes may be broken, pointer(s) referring to node(s) in the chain may become invalid, and so on. Unfortunately, a test case that causes such a defect to be executed but fails to follow it with the proper combination of additional method calls may fail to produce any observable difference in output. Because the BIT wrappers used in this experiment performed consistency checks on the internal state of the wrapped component, such corruption was detected as soon as it occurred. All of the additional defects detected by BIT wrappers that were not detected by observable output in this experiment were of this nature. This also explains why the magnification effect was greatest when testing was weakest—as test suites become more sophisticated and cover more combinations of methods in individual test cases, they are more likely to reveal such a defect; using a BIT wrapper, on the other hand, identifies the corruption as soon as it occurs, even in simpler test suites.

The preliminary results provided by this experiment are promising, although difficult to generalize. A more complete presentation of the experiment, its results, and threats to its validity and generality appears in [6]. However, these results, together with experience using the generator, indicate that there is the potential for practical automation of this testing strategy.

5 Related Work

The BIT wrappers here are built on a philosophy perhaps best phrased by Bertrand Meyer as design-by-contract [11]: preconditions of operations are the responsibility of callers while postconditions are the obligations of implementers, and implementers may thus assume that the preconditions hold at the time of invocation. Others have proposed different allocations of responsibilities [10, 14]. One key difference in the approach advocated here is that responsibility for checking whether or not obligations are met should be separated from both client and implementer. In addition to decoupling checking code from both the client and the component, this also opens up the opportunity of performing checks in client-level, abstract terms instead of in component-level implementation details. This results in highly reusable wrappers that easily can be added to or removed from a system.

Many others have also discussed the idea of run-time assertion checking. The Annotation Pre-Processor described by Rosenblum [15] is a good example. However, such approaches typically do not distinguish between the abstract view of component state perceived by clients and the concrete, implementation details seen by implementers. In addition, such approaches are rarely integrated into an overall strategy for automated testing. Eiffel provides another wellknown approach for pre- and postcondition checking at runtime [11]. A more complete discussion of differences between BIT wrappers and Eiffel assertion checking is provided in [5], but the Eiffel approach is not combined with a systematic approach to producing test drivers or test data.

The flowgraph-based test data generation research summarized in Section 4.3 is related to a large body of prior work, including DAISTS [7] and ASTOOT [4]. A more complete discussion of prior work in this area appears in [6].

Other published approaches to specification-based testing of object-based and procedural software components [2, 4, 7, 8, 13] have influenced this work. The research described here differs, however, in the way it incorporates run-time interface violation checking, a strategy for generating test data, a design for unit and integration test drivers, and the way it separates testing infrastructure code completely from all units under test in a system.

6 Conclusions and Future Work

This paper briefly sketches a general strategy for automated black-box testing of software components. The strategy is based on combining three techniques: automatic generation of component test drivers, automatic generation of test data, and automatic or semi-automatic generation of wrappers serving the role of test oracles. This research in progress unifies several threads of testing research into a coherent whole. Several difficult research questions remain open, but work to date indicates that practical levels of testing automation are possible.

The primary goals for future work include:

- Applying the technique on a larger scale to more realistic examples.
- Evaluating the costs and the benefits of this testing approach relative to existing testing techniques, both more traditional manual approaches and alternative automated approaches based on formal behavioral specifications.
- Evaluating the effectiveness of the test data generation approach more comprehensively.
- Empirically evaluating alternative heuristics, both for generating flow graphs from specifications and for selecting specific data values to be used in generated test cases.
- Experimentally assessing the overhead incurred by BIT wrappers.
- Exploring the practicality of including nonfunctional properties, such as time or space utilization in a suitable specification framework, within an extension of this approach.
- Exploring the limits of semi-automatic generation of postcondition checking code in BIT wrappers.
- Assessing the feasibility of dynamic verification of postconditions as an alternative implementation strategy for BIT wrappers.
- Completing and evaluating the test driver generator.
- Developing and evaluating an end-to-end automation tool based on these efforts.

Acknowledgements

The ideas and feedback provided by members of the Reusable Software Research Group at The Ohio State University and at West Virginia University have helped shape and direct this research. In addition, several graduates students have contributed to the exploratory work described here: Vinay Annojjula, Sharmin Banu, Nikhil Bobde, Duxing Cai, Didem Durmaz, Rajat

Gupta, Bob Hall, Mandar Joshi, Hunter Nuttal, Jason Snook, Kent Swartz, Manoj Thopcherneni, and Wei Wang. Their contribution is gratefully acknowledged.

References

1. Beizer B. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* Wiley: New York, 1995.
2. Bennett B, Sitaraman M. Validation of results in testing abstract data types: A method for automation. In *Proc. 1st Int'l Conf. Software Quality*, Dayton, Ohio, Oct. 1991.
3. Coplien JO. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley: Reading, MA, 1992.
4. Doong R-K, Frankl PG. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Software Engineering Methodology,* 1994; **3**(2): 101-130.
5. Edwards S, Shakir G, Sitaraman M, Weide BW, Hollingsworth J. A framework for detecting interface violations in component-based software. In *Proc. 5th Int'l Conf. Software Reuse*, IEEE CS Press: Los Alamitos, CA,1998, pp. 46-55.
6. Edwards SH. Black-box testing using flowgraphs: An experimental assessment of effectiveness and automation potential. *Software Testing, Verification and Reliability*, Dec. 2000; **10**(4), pp. 249-262.
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. Hoffman D, Strooper P. The test-graphs metholodogy: Automated testing of classes. *J. Object-Oriented Programming*, Nov./Dec. 1995; **8**(7): 35-41.
9. King KN, Offutt J. A FORTRAN language system for mutation-based software testing. *Software Practice and Experience*, Jul. 1991, **21**(7): 686-718.
10. Liskov B, Guttag J. *Abstraction and Specification in Program Development.* McGraw-Hill: New York, 1986.
11. Meyer B. *Object-Oriented Software Construction, 2nd Edition.* Prentice Hall PTR: Upper Saddle River, New Jersey, 1997.
12. 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.
13. Parrish A, Cordes D. Applying conventional unit testing techniques to abstract data type operations. *Int'l J. Software Eng. and Knowledge Eng.*, Mar. 1994; **4**(1): 103-122.
14. Perry DE. The Inscape environment. In *Proc. 11th Intl. Conf. On Software Eng.* IEEE CS Press: Los Alamitos, CA, 1989, pp. 2-12.
15. Rosenblum DS. A practical approach to programming with assertions. *IEEE Trans. Software Eng.*, Jan. 1995; **21**(1): 19-31.
16. Shakir G. *A Systematic Generator for Detecting Interface Violations in Component-Based Software*. M.S. Report, Dept. of Computer Science and Elec. Engineering, West Virginia Univ., Morgantown, WV, 1999.
17. Sitaraman M, Welch LR, Harms DE. On specification of reusable software components. *Int'l J. Software Eng. and Knowledge Eng.*, 1993; **3**(2): 207-229.
18. Sitaraman M, Weide BW, eds. Component-based software engineering using RESOLVE. *ACM SIGSOFT Software Enigineering Notes*, 1994, **19**(4): 21-67.

19. Wang C, Musser DR. Dynamic verification of C++ generic algorithms. *IEEE Trans. Software Eng.*, May 1997; **23**(5): 314-323.
20. Wing JM. A specifier's introduction to formal methods. *IEEE Computer*, Sept. 1990; **29**(9): 8-24.
21. Zhu H, Hall PAV, May JHR. Software unit test coverage and adequacy. *ACM Computing Surveys*, Dec. 1997, **29**(4): 366-427.
22. 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.