Designing a Programming Language to Provide Automated Self-testing for Formally Specified Software Components

Roy Patrick Tan
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
rtan@vt.edu

Stephen H. Edwards
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
edwards@cs.vt.edu

1. INTRODUCTION

Writing software is an error-prone activity. Compilers help detect some of these errors: syntactic mistakes plus those semantic mistakes that can be detected through the type system. However, locating faults beyond those detectable by the compiler (and other static analysis tools) is often relegated to the programmer, who must write thorough tests to ensure confidence in the correctness of the software.

Although the specification and verification community has traditionally focused on decreasing software bugs by static verification, research has increasingly explored the dynamic analysis of the conformance of software components to its specifications. That is, researchers are investigating systems that can tell us whether a program's behavior is consistent with its specification while the program is being executed. While dynamic techniques do not offer the same degree of assurance as full static verification, they may provide useful pragmatic benefits without the human intervention needed by currrent generation verification tools. When interpreted as a testing technique, dynamic analysis offers us a glimpse of future testing tools that offer another line of automatic error detection that augments the compiler, and helps the programmer reduce the number of tests he has to write.

Modern unit testing tools such as JUnit allow some automation of the testing process. Specifically, they allow the automated execution of tests. The job of writing tests remains the responsibility of the programmer. In writing a test for a software component, the programmer must `a.` exercise a component such that a bug is likely to manifest; and `b.` write code to detect the bug.

Current research suggests that the use of formal specifications, coupled with the right infrastructure, may alleviate much of the tedious process of writing the tests. For example, JML-JUnit [?] removes the need to write code that detects a component failure. It can act as a test-oracle by checking Java classes against their specifications as the test cases are being executed. While JML-Junit suggests an ideal strategy for combining test execution with specification-based oracles, an appropriate test case generation strategy is necessary for effective performance [?, Tan04]

Not surprisingly, formal specifications can also play a role in automatically generating the test-cases themselves. Korat [?], for example, can quickly generate linked data structures for inputs by using an invariant checker to filter out impossible structures. A method to automatically generate test-cases proposed by one of the authors also leverages the runtime-checking of specifications [?].

The possibility of combining these supporting techniques into a unified approach to dynamic verification leads to a new question: what form would such a consolidated testing tool take? Based on the concept that the best tool is one that is so transparent it is invisible, we envision infusing the necessary infrastructure directly within the programming language itself. A language that provides the necessary support for formal behavioral description could generate a compiled component that has the ability to execute and report the results of tests it has created for itself. This would amount to a built-in self-testing capability that comes for free, as a side effect of writing formal specifications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAVCBS '04

2. A VISION FOR A NEW STYLE OF PROGRAMMING LANGUAGE

We envision that in the future, when a software component is ready for testing, it will have a formal specification written for it. Ideally, the specification would be complete, correct, and written well before the implementation. More probably, the specification might have missing parts, incorrect parts, and would have evolved as the implementation was written.

In any case, when the developer is ready to test his component, he runs his testing tool, and the tool will automatically generate the test-cases, run them, determine which tests passed or failed, and generate a report. This report will tell the programmer which parts of the component it found to be inconsistent with its specification. Depending on the report, the software engineer may fix some of his implementation code; he may refine his specifications; or he may write additional tests that the automated tool does not cover. The process iterates until the developer is confident

```
    public void testPush() {
        IntStack stack = new IntStack();
        stack.push(5);
        Assert.assertTrue(stack.size() == 1);
        Assert.assertTrue(stack.top() == 5);
    }
```

Figure 1: A JUnit test case method for an integer stack.

that the component works as specified.

In this scenario, the software developer writes much fewer test cases, though he is not completely rid of writing them. Just like most modern programmers do not normally need to bother with low-level details such as register allocation, programmers of the future will not have to write lower-level test cases. Instead, the programmer may concentrate on writing test cases for more subtle, hard-to-find bugs.

The programmer here never has to write any code to determine the correct behavior of the component under test. Instead, he has to write formal specifications. Making the programmer write specifications may be the most difficult part of transitioning from the current way of writing software. However, this may be mitigated in part by the fact that the techniques we are considering to bring us closer to this vision do not require complete or comprehensive specifications.

3. TECHNIQUES FOR AUTOMATED UNIT TESTING

Much of the ground work for our scenario for the future of unit testing has already been done. We believe it is possible to automate at least partially the two things a developer has to do manually in testing: exercise the component, and detect a fault if it occurs. The apporoaches we have been looking at have these two key aspects: software components have to be specified formally, and that there is a runtime environment that executes these specifications alongside the implementation. That is, specifications such as preconditions, postconditions, and class invariants must be checkable at runtime, whenever a method is called—a design-by-contract style of specification execution.

3.1 Using Specifications as a Test-Oracle

Figure 1 is an example of what a JUnit test case method for a stack may look like. Take note the two `assertTrue` calls, these assertions tells JUnit what must be true after the push statement. Every test of push has to have "assertions" similar to the one in Figure 1. It would be advantageous if we could write assertions in a single location that tells JUnit what must be true after every call to the push method.

Using the Java Modeling Language (JML) [?], to specify Java classes, you can do exactly this. The commented parts of Figure 2 is the postcondition of the `push` method. Since JML can execute the postcondition every time the push method is called, there is no more need to write assertions for the push test. Instead we can let JML detect the fault for us.

This is in fact what the JML-JUnit tool does; it uses JML's runtime checking of specifications as a test oracle. Thus, with formal specifications and the right runtime infrastructure, specifications can be used to check correct behavior in lieu of manual assertions inserted in test cases.

```
    //@ensures size() == \old(size()) + 1
    //@        && top() == x;
    public void push(int x) {
        //...
    }
```

Figure 2: A partial JML specification of the push method.

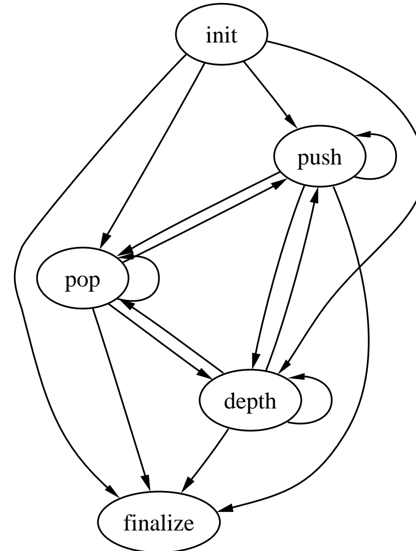

Figure 3: A flow graph for a stack component

3.2 Test cases generation

In [?], one of the authors (Edwards) presents a strategy of generating test cases using flow graphs which in turn is based on the methodology described by Zweben and Heym [?]. We present a brief explanation:

Given a specified component, we build a graph where any walk represents a possible object lifetime. We define a flowgraph as follows:

> A flowgraph is a directed graph where each vertex represents one operation provided by the component and a directed edge from vertex v_1 to v_2 indicates the possibility that control may flow from v_1 to v_2.[?]

In other words, when there is an edge from v_1 to v_2, it means that there exists an object state where v_2 can be legally called after v_1. A flowgraph for any component can be constructed in the following way: Represent every method as a node in a graph. Construct a complete, directed graph with self-loops from these vertices. And then, add two more nodes, *begin* and *end*; place a directed edge from *begin* to every node and from every node to *end*. Additionally, there is an edge going from *begin* to *end*. Figure 3, for example, is a flowgraph for a stack component.

Thus, a walk from the *begin* vertex to the *end* vertex represents a sequence of method calls from object initialization to object finalization, i.e. a possible object lifetime. It is easy to see that each feasible walk can be a test-case for the component.

There are two problems that come to mind, one is that some of the walks may be infeasible. For example, the sequence of method calls represented by $begin \rightarrow push \rightarrow pop \rightarrow pop$ for a stack component may be infeasible, because the last pop call violates the method's precondition. The other problem is that there are a pontentially infinite number of feasible walks through the graph.

The problem of infeasible walks can be solved by using the dynamic execution of specifications to detect them. An infeasible path is detected when a precondition failure occurs while executing a sequence of method calls represented by a walk on the flowgraph.

The second problem of choosing the right walks to use as test-cases is an open topic for research. There are several possible ways to achieve this:

- Random walk. Random walks are simple to implement but may not be best.

- Bounded exhaustive enumeration. For example, choose all walks going through 5 nodes or less.

- Various machine-learning algorithms. Tonella [**?**], for example, reports on an experiment that uses evolutionary algorithms to essentially generate these walks.

Work is ongoing to investigate the efficacy of each of these strategies.

4. SUPPORTING AUTOMATED TESTING

In the previous section, we see that a software developer who is willing to write formal specifications may be able to take advantage of a higher level of automated testing. Aside from the developer's willingness to write the formal specifications, however, the developer must possess tools that can take advantage of these specifications.

What are the necessary requirements to be able to build these tools? What language features must exist for our automated testing strategy to work? The basic necessities are that the programming language must have its components formally specifiable, and that there is a runtime system that can execute the specifications in a design-by-contract style.

We believe that any language with design-by contract style specifications (and the ability to check them at runtime) is amenable to the automated testing strategy we outline above. However, we have also listed a number of secondary characteristics that may be beneficial:

- Simple specification language—a simple language allows for easier programmer buy-in, part of this is to have the specification language be as close as possible to the implementation language, to make it easier to learn.

- Support for modular reasoning—modular reasoning means that each module (e.g. a class) is as encapsulated as possible; that it can be reasoned about in isolation of the rest of the program, and thus can be tested in isolation.

- Small programming language—a small language without too many features may make for simpler specification.

- Ability to measure other metrics—such as time for every method call, code coverage, etc.

Several programming languages have the necessary characteristics to implement tools that follow our testing strategy. The aforementioned JML-JUnit tool, for example, already uses runtime checkable specifications as a test oracle. Eiffel, which popularized design by contract, is certainly a candidate for this type of tool. Theoretically, design-by-contract extensions to popular scripting languages such as Python [**?**] and Ruby can also be used.

Each of these languages, however, also have characteristics that makes building automated testing tools for them difficult. For example, Java allows direct access of data members, breaking modularity of reasoning. The meta-programming features of Python and Ruby, might allow developers to circumvent specification checking. Eiffel breaks the Liskov substitution principle [**?**] All the languages considered are also fairly feature-rich; building a tool that covers all the features of one of these languages may be beyond the resources of academic researchers. The use of reference semantics in all these languages introduce aliasing, which also introduces all the difficulties associated with specifying them.

We have decided to take on the challenge of designing a new programming language and its runtime system. Tentatively called Sulu, we are designing it with the goal that every component written in this language can be tested automatically. By implementing a new language we will have the advantage of having complete control of the language, we can make it only as large as necessary, place only the features we require. It can also serve as a platform for future research.

5. DISCUSSION

We discuss many of the technical concepts of building the automated testing tools that we envision. But beyond building the tools, we must be able to measure their effectiveness. We must also measure other metrics like the number of test cases, the time it takes for the automated process to generate them, and the time it takes to execute the tests.

By deciding to implement a new programming language, we encounter a new set of challenges; what features should we put in the new language? What should be left out? We must strike a balance between making it small enough to be easy to implement, and big enough to show that our techniques are also applicable to mainstream languages.

The key elements of a specification language and the ability to check the specifications against the implementation at runtime will be included, of course, but what about other features? Sulu will be component-based. That is, it will have strong separation of an object's specification and its implementation. It will use value semantics to avoid the difficulties of specifying aliased variables. Performance concerns will be addressed somewhat by allowing a swap operator [**?**].

We are still actively evaluating whether to include other features, such as the object-oriented concepts of inheritance and polymorphism. These features may make results from

future experiments more comparable to mainstream languages, but it may also mean a much more complicated implementation and specification language.

Another crucial question that may need to be addressed is the cost/benefit to the software developer. Will the promise of automated test-case generation convince practitioners to write formal specifications? How effective should the tools be to facilitate this change?

If the developer does write formal specifications, this artifact may be useful for other analysis tools. How can traditional verification tools be used in conjuction with the testing tools to help us build better, more reliable software?

6. CONCLUSION AND RELATED WORK

In this paper, we have outlined our vision for unit testing, that testing tools will come to the fore as another level of automatic error detection.

We have discussed the techniques we are implementing as we develop our testing platform, but there is a fair amount of other research on the automatic generation of test cases. ASTOOT [**?**] and DAISTS [**?**] approach the problem quite differently. They automatically generate test cases directly from algebraic specifications through term-rewriting.

Korat [**?**] is a system that automatically generates linked data structures that can be used as parameters for methods under test. Korat's use of a "representation invariant" to filter out infeasible data structures is similar and may be compatible with our technique.

Tonella [**?**] describes a system in Java that uses an evolutionary algorithm to generate method sequences essentially equivalent to walks in our flowgraph model. However, he does not consider the problem of infeasible method sequences.

The basic techniques to achieve this vision already exist, and discussed how we can support our techniques in a modern programming language.